Advanced C Programming

Student Workbook

itcourseware
THE SOURCE FOR IT PROFESSIONALS

Advanced C Programming

Jeff Howell

Published by itcourseware, 10333 E. Dry Creek Rd., Suite 150, Englewood, CO 80112

Special thanks to: Many instructors whose ideas and careful review have contributed to the quality of this workbook, including Brandon Caldwell, Denise Geller, Todd Gibson, Roger Jones, Channing Lovely, and Danielle Waleri, and the many students who have offered comments, suggestions, criticisms, and insights.

Contents

Chapter 1 - Course Introduction

Notes

Course Objectives

- Continue development of C programming abilities by writing a variety of C programs.
- Deepen understanding of variable attributes such as local, global, external and static.
- Develop proficiency in writing programs that perform file and interactive I/O.
- Gain an understanding of how C implements data structures.
- Write small scale applications that “brings together” the newly gained knowledge and skills.

Notes

Course Overview

- **Audience:** This course is for intermediate C programmers who need to design, implement, debug, and test C programs of varying complexity.
- **Prerequisites:** A beginning course in C. This course does not teach basic C programming. It is meant as a more advanced course for people who have basic C programming skills.
- **Student Materials:**
  - Student Workbook.
- **Classroom Environment:**
  - Individual Terminals.
  - Reference Materials.

Notes

Suggested References

Kelley, A., and Pohl, I. 1998. ***A Book on C, Fourth Edition***. Addison Wesley, Reading, MA. ISBN 0-201-18399-4.

Kernighan, B., and Ritchie, D. 1988. ***The C Programming Language,* Second Edition**. Prentice Hall, Englewood Cliffs, NJ. ISBN 0-13-110362-8.

Kruse, R., Leung, B. and Tondo, C. 1996. ***Data Structures and Program Design in C***. Prentice Hall, Englewood Cliffs, NJ. ISBN 0-13-288366-X.

Kumar, R., and Agrawal, R. 1992. ***Programming in ANSI C***. West Publishing Co., St. Paul, MN. ISBN 0-314-89563-9.

Stevens, W. 1992. ***Advanced Programming in the UNIX Environment***. Addison Wesley, Reading, MA. ISBN 0-201-56317-7.

Notes

Chapter 2 - The C Development Environment

Notes

Chapter Objectives

- Explain what happens at the `cc(1)` command.
- Understand the difference between include and library files.
- Create a library of object files.
- Link with external libraries.
- Optimize executable size.
- Debug ‘undefined symbol’ errors.
- Create robust code using `cc(1)` command line options.

Notes

The `cc(1)` command automatically invokes the four phases described on the following page. Options to the `cc(1)` command allow you to stop the compilation after completing any particular phase. Use the `-V` option to display the phases of the compile process.

As an example, create the (infamous) program `hello_world.c`. Compile `hello_world.c` with the `-V` option to `cc(1)`:

```
$ cc -V hello_world.c
```

Now convert `hello_world.c` to a function, then create `hello_main.c` which calls the function in `hello_world.c`. Compile `hello_main.c`. What happened? What additional information must be supplied to `cc(1)`?

INVESTIGATE:

Read the man pages to see other options available to `cc(1)`. What option is used to pre-process the source? Which option compiles but doesn't assemble the code? What are the names of their corresponding output files?

The cc (1) Command

- `cc(1)` is typically a “driver” program.
- `cc(1)` executes the following programs:

  1.) C pre-processor (cpp)
  2.) C compiler (acomp or ccom)
  3.) Assembler (as)
  4.) Linker (ld)

- The C pre-processor looks for pre-processing directives to the compiler.
  - Syntax errors will not be caught yet.
  - The output is typically placed in a file with suffix `.i`.
  - What are examples of C statements processed in this phase?
- The C compiler translates the pre-processed source into assembler code.
  - This phase of the compiler catches syntax errors.
  - The output is typically placed in a file with suffix `.s`.
- The assembler turns the compiled source into a binary object file.
  - The resulting `.o` file is normally deleted if a single C source file is compiled and then immediately linked.
  - What option to `cc(1)` would you use to keep the `.o` file? (We will see many instances of why to keep the object file.)
- The link editor combines object files to produce an executable file.
  - The `ld(1)` command resolves external references.
  - You may also specify library files to search for objects. (We will discuss libraries soon).
  - The resulting executable, by default, is named `a.out`.
  - What option to `cc(1)` is used to override the default executable name?

Notes

Your code will be more portable and robust if you use command line options to specify pathnames to header files rather than hardcoding them in your source. Command line options are usually placed within makefiles.

For example, to include specified header files that exist in directories `/home/denise/mydir` and `/home/fred/theirdir`, use the following command line:

```
cc fn.c -o fn -I/home/denise/mydir -I /home/fred/theirdir
```

To further clarify what an include file is, change the function `hello_world` to return the value of pi (3.14). What modifications did you make to the function? Create the corresponding header file, `hello_world.h`.

Add the following pre-processing directive to `hello_main.c`:

```
#include <hello_world.h>
```

Also, add the following statements to `hello_main.c`:

```
float pi;

pi = hello_world();
printf ("Have a piece of %f\n", pi);
```

Compile `hello_main.c`. What happened? Could the compiler find your header file? Try compiling with the command:

```
$ cc hello_main.c -I.
```

What happened? What error message did you get? On which phase of the compile process did it fail? What additional information must be supplied? Why?

Include Files

- Include (header) files contain C source statements that are read in during the pre-processing phase of the compiler.
  - They contain declarations (what).
  - Header files are customarily suffixed by `.h`.
  - What are examples of things to place in header files?
  - What are examples of things not to place in header files? Why?
- On UNIX, standard header files included with the C environment are found in `/usr/include`.
- The angle brackets in `#include <stdio.h>` tell the compiler to search for the file `stdio.h` in the directory `/usr/include/`.
- The double quotes in `#include "myheader.h"` tell the compiler to search for the file `myheader.h` in the same directory as the source.
- The `-I` option to `cc(1)` tells the compiler to also look for header files in the directory paths specified on the command line.
  - This option works only for header files specified with a relative pathname. Absolute pathnames override the search paths specified by `-I`.
  - The compiler will search these directories (in the order presented) prior to searching the usual directories for header files.
  - Repeat the `-I` option for each directory to be searched.
  - If more than one directory is specified, searching stops at the first directory containing the specific header file.
  - Either notation (quotes or brackets) may be used with the `-I` option.

Notes

When writing code, you will often create functions that others will want to use. The compiled (and presumably tested) functions can be placed into a library accessible to other programmers. The Standard C library contains hundreds of object files, each representing a separate function. These previously created functions have already saved you vast amounts of time in your code development. How much longer would it take you to develop code if you could not access the existing I/O or math routines?

Libraries act as depositories for object modules (`*.o`). Which objects should be placed in libraries? Any functions that may be called by multiple programs should be compiled separately, thoroughly tested, and their objects placed within a library. Anyone reusing that function will be implicitly "reusing" the associated testing, saving time from having to retest that function.

In UNIX, the `ar(1)` command is used to create libraries. The `ld(1)` command will access libraries in attempting to resolve remaining external references. If the function is found in a library, only that function is linked into the executable file. This reduces executable size since only what you need is loaded.

To list the contents of the Standard C library: `$ ar t /usr/ccs/lib/libc.a`

How would you list the contents of the math library?

The steps to create a library are :

1) Create and test `hello_world.c`.
2) Place the function prototype in a (new) header file.
3) Create the object to be archived:
```
$ cc -c hello_world.c.
```
4) Archive `hello_world.o.`
```
$ ar rv  /home/denise/libmine.a hello_world.o
```
5) Remove `hello_world.o` from your source directory.
```
$ rm hello_world.o
```

The `rv` keys to `ar(1)` tell it to create the library if it does not already exist. If it did exist, then create or replace the `hello_world.o` file in the library. Use `v` for verbose mode. How would you delete a file from the archive?

To link `hello_world.o` in a program:
```
$ cc new_prog.c -L /home/denise -I /home/denise -l mine
```

Libraries

- Libraries contain object (`*.o`) files that are searched during the link phase of the compiler.
  - They contain the function definitions (how).
  - Libraries are searched to resolve external references.
- The standard C Library, `libc.a`, is automatically loaded by the `cc(1)` command when it calls `ld(1)`.
  - Functions listed within sections 2, 3C and 3S of the man pages constitute the standard C library.
- Other function libraries to be searched must be specified to `ld(1)`.
- "undefined symbol" errors are solved by linking with additional object files, often stored within a library.
- The `-l` option to `cc(1)` tells the compiler to look for object files in the libraries specified on the command line.
  - This option and its argument are passed to `ld(1)`.
  - Repeat the `-l` option for each library to be searched.
  - The compiler will search the specified libraries in the order presented.
  - Searching stops when a reference is resolved.
- The command `$ cc fn.c -lx` tells the linker to search `libx.a`.
  - By convention, UNIX libraries are named `libname.a`.
  - The `ld(1)` command will look for unresolved symbols in `libx.a` before searching the standard C library.
- Searches other than in the default library directories must be specified with the `-L` option.
  - The directories specified with `-L` are searched before the Standard C library.
  - This option and its argument are passed to `ld(1)`.
  - Repeat the `-L` option for each path to be searched.
- Any remaining unresolved references result in an "undefined symbol" error.

Notes

Exercises

1) Create the header file "joke.h". Include the following 2 lines inside joke.h:

```
printf ("To get to the other side.\n");
#define ANSWER 42
```

Now create the program "joke_main.c" as:

```
void main (void)
{
  printf ("Why did the chicken cross the road?\n");
  #include "joke.h"
  printf
  ("The meaning to life, the universe and everything is %d.\n",ANSWER);
}
```

Compile and execute joke_main. What affect does the #include statement have? When does the named number ANSWER get its value? What option to the cc command can you use to verify this? Try it and look at the resulting file.

2) What are the primary differences between an include file and a library? Why don't we have to compile include files separately?

3) What are the advantages of using separate compilation and placing commonly used functions alone in their own .c source files?

4) Place the previously created function hello_world into a library, then remove the object file from your current directory. Recompile and link the hello_main program using your new library. What options to cc did you use?

OPTIONAL

A) Try to create the executable for joke_main.c by manually invoking each of the four separate phases of the cc command. Were you able to generate an a.out? What was the error message produced when you tried to link the object file?

What library must you include manually with the ld command that the cc command automatically links in for you?

What happened when you ran a.out? Investigate the man pages for cc to see what additional files must be linked, then execute the resulting program again.

Notes

Chapter 5 - Standard I/O

Notes

Chapter Objectives

- Create, access and close files using the standard I/O routines.
- Redirect I/O using the operating system.
- Read and write formatted I/O using `fscanf()`.
- Write robust code by using `fgets()` instead of `gets()`.
- Read and write structures to a file using block I/O.
- Update data in a file using random access.

Notes

Three streams are opened automatically at the start of every C program. Their names and associated `FILE` pointers are: standard input (`stdin`), standard output (`stdout`) and standard error (`stderr`). These are defined in `<stdio.h>` and, by default, are connected to your terminal.

`stdin` is where functions like `gets()` and `scanf()` read their input from. `stdout` is where functions like `printf` and `puts()` write their output to. `stderr` is where your error messages go, which may or may not be the same place as your standard output.

Programs that read from `stdin` and write to `stdout/stderr` can have their streams redirected by the shell.

`stdin` and `stdout` are buffered: characters are not written to the device until a newline character is encountered. `stderr` is never buffered.

Standard I/O Streams

- A stream is a file pointer that is used to uniquely identify an open file.
- I/O on files consists of 3 basic steps :

  1) Establish a stream on file open
  2) Perform I/O on file
  3) Close the file

- When opening a file for I/O, the programmer will establish a unique stream (`FILE *`) to associate with the file.
  - This provides the “connection” between the program and the data file.
  - All subsequent operations on the file are conducted through the file pointer.

Notes

There are 6 modes to select from on a file open:

- `r` - allows read only on file
- `r+` - allows read and write
- `w` - allows write only to file, truncates existing files
- `w+` - allows read and write to file, truncates existing files
- `a` - allows append to file
- `a+` - allows append to and read from file

`r, r+` will require that the file already exist
`w, w+` will create a new file by the specified name
`a, a+` will allow append to end of existing files, and will create the file if not existing
(modes specified with `+` are used for update mode: both reading and writing)

Additionally, you may append a "`b`" (e.g.: "`wb+`") to the above mode to specify to the system to use a binary (instead of a text) stream. UNIX treats both binary and text files the same, making the appended "`b`" uneccessary.

Another useful function is `perror()` (in `<errno.h>`) whose prototype is:

```
void perror (char *string).
```

The externally defined `errno` is automatically set by most library routines when an error is encountered. `perror()` prints `string`, followed by the error text associated with `errno`. If the routine terminated with no error, the previous `errno` value is left unchanged, so be sure to check that the library routine failed before using `perror()`.

File Access

```
#include <stdio.h>
FILE * fopen (const char *fname, const char *mode);
int fclose (FILE *fp);
```

- `fopen()` is used to open a stream for reading or writing.
  - `fname` is the name of the file.
  - `mode` describes how the file is to be used.
- A file pointer is returned, pointing to the `FILE` structure defined in `<stdio.h>`.
  - The `FILE` structure contains information about the file such as the current character position and the file mode.
  - `NULL` is returned if the `fopen()` fails.
- `fclose()` closes the associated stream.
  - Any buffered data gets written out.
  - Returns 0 if successful, or `EOF` if there were any errors.
- For example:

```
...
FILE *fp;
/* open existing file for update */
if ( NULL == (fp = fopen ("filename", "r+")))
{
   /* if no such file, then create */
   if ( NULL == (fp = fopen ("filename", "w+")))
   {
      /* fatal error: perhaps permissions problem? */
      perror ("filename");
      exit (errno);
   }
}
...
fclose (fp);
```

Notes

The `scanf()` function is particular about the format string arguments. Embedded blanks in the format string are ignored when using numeric or string conversions. On input, leading white space on numeric or string arguments is skipped. For example:

Using input containing "`16ducks 42.000000`", the following program:

```
#include <stdio.h>
void main(void)
{
   FILE * fp;
   int int_x, int_x2;
   char str_var2 [6];
   float flt_y2;
   fp = fopen ("ex2.out", "r");
   fscanf (fp, "%d      %s %f", &int_x2, str_var2, &flt_y2);
   fprintf (stdout, "%d %s %4.1f\n", int_x2, str_var2, flt_y2);
}
```

produces the output: `16 ducks 42.0`

But when using character format conversion ("`%c`"), white space becomes significant (the normal skip over white space characters is suppressed). For example, using input containing `A two 22`, the following program:

```
#include <stdio.h>
void main(void )
{
   char ch_one, ch_two, ch_three, ch_four, ch_five, ch_six;
   FILE *fp;
   fp = fopen ("ex3", "r");
   fscanf (fp, "%c%c%c%c%c%c",
      &ch_one, &ch_two, &ch_three, &ch_four, &ch_five, &ch_six);
   fprintf (stdout, "%c%c%c%c%c%c\n",
      ch_one, ch_two, ch_three, ch_four, ch_five, ch_six);
}
```

produces the output: `A two`

Inserting a blank within the format argument string will force leading white space to be ignored. In the example above, changing the format string in `fscanf()` from "`%c%c%c%c%c%c`" to "`%c %c %c %c %c %c`" produces the output: `Atwo22`

Formatted I/O

```
#include <stdio.h>
int fprintf (FILE *fp, const char *format, ...)
int fscanf (FILE *fp, const char *format, ...)
```

- `fprintf()` is similar to `printf()`.
  - `fprintf()` converts, formats and prints its arguments into `fp` using `format`'s descriptions of any subsequent arguments.
  - It returns the number of converted and printed values or a negative number on error.
  - `printf(...)` can be implemented as `fprintf(stdout, ...)`.

- For example:
  `fprintf (fp, "%d %s %f", int_x, str_var, flt_y);`

- `fscanf()` is similar to `scanf()`.
  - Each argument of `fscanf()` must be a pointer.

- Input fields are a string of non-white space characters, extending to either the next white-space character or until a specified field width is met.

- `fscanf()` is the input equivalent of `fprintf()`.
  - `fscanf()` converts, formats and reads its values from `fp` into the subsequent arguments using format's descriptions.
  - It returns the number of converted and assigned values, or EOF on error.
  - `fscanf()` is often used in conjunction with `fprintf()`.

- For example:
  - `fscanf (fp, "%d %7s %f", &int_x, str_var, &flt_y);` reads and converts input into integer, string and floating point variables.
  - `fscanf (fp, "%*d %7s %f", str_var, &flt_y);` will skip a preceding integer value, continuing with the next string.

Notes

In C, you are responsible for protecting your array bounds. The compiler will not catch overstepped bounds. The following program:

```
#include <stdio.h>
void main(void)
{
   int a = 1, b = 2;
   char str[3];
   printf ("Enter a string ");
   gets (str);
   printf ("The value of a is %d b is %d and str is %s\n",a,b,str);
}
```

produced: `Enter a string hi`
`The value of a is 1, b is 2 and str is hi`

But running the script a second time produces the following:

```
Enter a string this is a long string
The value of a is 1814061344 b is 1936269427 and str is this is a
long string
Memory fault(coredump)
```

Too many characters were entered into the string variable `str`. (We will see what happened in memory in a later chapter.) You can protect your array bounds by using `fgets()`, which will allow a maximum of `limit-1` characters to be read. Changing `gets(str);` above to `fgets (str, 3, stdin);`

produced: `Enter a string this is a long string.`
`The value of a is 1, b is 2 and str is th`

String I/O

```
#include <stdio.h>
int fputs(const char *string, FILE *fp);
char *fgets(char *string, int limit, FILE *fp);
```

- Use `fgets()` in combination with `fputs()`.
- `fputs()` is similar to `puts()`.
  - `fputs()`, unlike `puts()`, will not automatically append a newline character.
  - Returns a non-negative number, or EOF if an error occurs.
- For example:

```
fputs("This string will be input to the file\n", fp);
```

- `fgets()` is similar to `gets()`.
  - Reads until `limit-1` characters are read, or a newline character is read.
  - The nth character will be `\0`.
  - Returns string, or NULL on error or end of file.
  - `fgets()`, unlike `gets()`, will not automatically strip off a newline character.
- Using `fgets()` is safer than using `gets()`.

Notes

As an example of using file positioning operations, this program finds the misspelled word "`decypher`", and replaces the incorrect `'y'` with the correct character `'i'`.

```
#include <stdio.h>
#include <errno.h>
void main (void)
{
   FILE *fp;
   char me [10];
   fpos_t pos;
   int position;
   if ( NULL == (fp = fopen ("filename", "r+")))
   {
      /* fatal error */
      perror ("filename");;
      exit (errno);
   }
   while (1)
   {  /* get current position */
      position = fgetpos (fp, &pos);
      /* scan to first white space */
      fscanf (fp, "%s",me);
      if (!feof(fp))     /* haven't yet read end of file */
      {
         if (strcmp (me, "decypher") == 0)
         {
            /* found the typo, set position back on preceding white space */
            fsetpos (fp, &pos);
            /* go to the erroneous character 'i' */
            fseek (fp, 4, SEEK_CUR);
            /* make the correction */
            fprintf (fp, "%c", 'i');
            /* read back the corrected word for display */
            fsetpos (fp, &pos);
            fscanf (fp, "%s",me);
         }
         fprintf (stdout, "%s ",me);
      }
      else
      {
         /* at end of file, put in the \n */
         fprintf (stdout, " \n ");
         break;
      }
   }
}
```

On the input file `filename` containing the string: "`The word decypher is spelled with an 'i'.`" The above program corrects the misspelled word decipher.

This program also introduces the function `feof()`, which returns non-zero if the previous read operation on the stream hit the end of file.

File Positioning Operations

```
#include <stdio.h>
long ftell (FILE *fp);
int fseek (FILE *fp, long offset, int start);
void rewind (FILE *fp);
int fgetpos (FILE *fp, fpos_t *position);
int fsetpos (FILE *fp, const fpos_t *position);
```

- You do not have to access each byte successively to obtain data from a file. File positioning operations allow random access to files.
- `ftell()` reports the current value of the file position indicator.
  - The returned value may be used as argument to `fseek()` to reset the file position indicator.
  - Returns `-1L` on error.
- `fseek()` sets the file position indicator for the next operation.
  - The new position is set at `offset` bytes from `start`.
  - `start` has values of beginning of file (`SEEK_SET`), current position in file (`SEEK_CUR`) and end of file (`SEEK_END`).
- On non-UNIX systems, `fseek()` and `ftell()` are guaranteed to work correctly only on binary files (append "`b`" onto `mode` in `fopen()`).
- `rewind()` places the file position indicator to the beginning of the file and clears error indicators for `fp`.
- `fgetpos()` is new to ANSI C.
  - Returns the current value of the file position indicator into `position`.
  - `position` may be used as an argument to `fsetpos()`.
  - Returns a non-zero if error.
- `fsetpos()` is new to ANSI C.
  - `position` is retrieved previously with `fgetpos()`.
  - Sets the file position indicator for the next operation at `position`.
  - Returns a non-zero if error.

Notes

`fread()` and `fwrite()` also read from and write to files. These functions are used to read and write unformatted data (ie: no conversions are performed). You can use these routines to write one object or an array of objects to a file. The example below illustrates both:

```
#include <stdlib.h>
#include <errno.h>
#include <stdio.h>
void main (void)
{
   FILE *fp;
   int nobj,i;
   int a[3] = {54, 55, 56};
   int b[4] = {0};   /* use to verify the write */

   /* open the file for update : both writing and reading */
   if ( NULL == (fp = fopen ("sfile", "w+")))
   {
      /* fatal error */
      perror ("filename");;
      exit (errno);
   }
   /* write second element of the array to the file */
   if ((nobj = fwrite (&a[1], sizeof (int), 1, fp)) < 1)
   {
      perror ("writearray.c *** 1");
      exit (EXIT_FAILURE);
   }
   /* now write the entire array to the file */
   fwrite (a, sizeof (int), (sizeof a)/sizeof(int), fp);

   /* must flush buffer before switching from write to read mode */
   rewind (fp);
   /* read the first element into b */
   if ((nobj = fread (&b[0], sizeof (int), 1, fp)) < 1)
   {
      perror ("writearray.c *** 3");
      exit (EXIT_FAILURE);
   }
   /* now read rest into remaining b array elements */
   fread (&b[1], (sizeof b)-sizeof (int), 1, fp);
   /* print results to verify I/O */
   for (i=0; i<4; i++)
      printf("b[%d] is %d\n",i,b[i]);
}
```

Block I/O

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t nobj, FILE *fp);
size_t fwrite (const void *ptr, size_t size, size_t nobj,
FILE *fp);
```

- `fread()` reads from stream `fp` into the array `ptr`.
  - At most `nobj` objects of `size` bytes are read into `ptr`.
  - The number of objects read is returned, which may be less than the number requested. If less than `nobj` objects are read, the end of file marker was probably encountered.
- `fwrite()` writes from the array `ptr` onto stream `fp`.
  - Writes at most `nobj` objects, each object being `size` bytes. If less than `nobj` objects are written, an error occurred.

Notes

By default, reading and writing is buffered (except to `stderr`). For example, rather than going directly to the device for every character on every `read`, the system reads in a buffer full of characters on the first read request. Further reads are taken from the buffered characters. A new buffer full of characters is retrieved when the buffer empties.

An explicit call to `fflush()` will write out any buffered data.

```
int fflush (FILE *fp);
```

A `0` value is returned if no errors occurred, otherwise it returns EOF.

`fflush` (`NULL`) flushes all opened output streams. `fflush (stdin)` eliminates remaining (unwanted) characters from the standard input stream.

When switching from read mode to write mode (or vice versa) on a file opened in update mode, the buffer must be flushed. Ways to accomplish this are:

1) close the file, then reopen the file.
2) explicity calling `fflush()`.
3) calling a file positioning function: `fseek()`, `fsetpos()` or `rewind()`.

Exercises

1) Standard input and standard output are automatically defined for a program by UNIX. You may redirect `stdin` and `stdout`. Create the following program `inp.c`:

```
#include <stdio.h>
void main (void)
{
     char s[80];
     printf ("The contents of the file follow : \n");
     while (gets (s) != NULL)
          puts (s);
}
```

After testing your program interactively, create the file `ex1_in` with the following text:

```
This is an example
file to display
redirection of input and output.
```

Run your program using the following command: `$ inp < ex1_in > ex1_out`

Describe what happened.

2) What will be output from the following program?

```
#include <stdio.h>
void main (void)
{
     fprintf (stdout, "C ");
     fprintf (stderr, "is ");
     fprintf (stdout, "fun.\n");
}
```

Enter the program and verify your answer.

Notes

Exercises (continued)

3) Enter and run the following (`buggy`) program that prompts the user to enter 2 characters, one at a time:

```
#include <stdio.h>
main()
{
      char ch1, ch2;
      printf ("Please enter one character ");
      fscanf (stdin, "%c", &ch1);
      printf ("Please enter one character ");
      fscanf (stdin, "%c", &ch2);

   printf ("You entered the following chars %c %c\n", ch1,
          ch2);
}
```

What happened when you ran the program? What do you think the `ch2` variable contains? To fix the program, study the `fflush` function and apply it in your program. Does it work now? What did the `fflush` function do?

Notes

Exercises (continued)

4) Enter the following program. This program uses the `fwrite` function to input the contents of the `iarr` array into the `moose` file.

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
    int iarr[] = {71,79,79,68,32,74,79,66,33,0};
   FILE *fp;
   int c,i;

    if ((fp = fopen ("moose", "w")) == NULL)
   {
        perror ("lab4");
        exit (EXIT_FAILURE);
   }

   fwrite (iarr, sizeof(int), (sizeof iarr)/sizeof(int),
fp);
}
```

What does `sizeof(iarr)/sizeof(int)` represent? Would this statement still apply within a function that you had sent in the array as a parameter?

After running the program, view the contents of the `moose` file.
What was entered into `moose` and why?

Optional

A) Create two additional functions for your employee program. The first function will write the array of employee data into a file. The second will read the employee data back into an array of employees. What parameters must you supply for the function prototypes? Change your program so that at program startup, it reads existing employees from a file into the array. At program end, write out the (new/changed) array of employees to the file.

Notes

Chapter 9 - Data Structures: Linked Lists

Notes

Chapter Objectives

- Explain self-referential data types.
- Determine when a linked list may be the best choice for a data structure in solving a programming problem.
- Design and write C programs to perform operations on linked lists.

Notes

Arrays are often suitable for holding constant amounts of information. However, your programs will often have to meet dynamic (constantly changing) conditions while processing data. How would an array of employees accommodate the following situations?

1) If your company should grow or shrink in size, you may have to change your code to adjust the array size.

2) If a new employee starts, you may have to place them in sorted order into your existing list. Those with names following will have to be pushed downward within the array.

3) Another employee wins the Lotto and resigns. Those following them in name order will have to be pushed up to fill the vacant spot in the array.

4) What if your program is to be used corporate wide to keep track of all 30,000 employees, while at the same time be used within the individual branches which may have between 25 and 5000 employees? How big should you dimension your array?

Problem - Array Limitations

- Up to now, you have been using arrays to maintain a list of information.
  - If your company has 10 employees, you might read their data into an array of 10 employee structures.
- Typically, you would store the information in a sorted array.
- Because arrays are fixed in size, they may not be the appropriate data structure for all situations.
  - Arrays are allocated statically (at compile time) and in most programming languages they cannot be re-sized during run time to meet changing needs.
- Maintaining the array can be cumbersome.

Notes

The structures in a linked list are typically called nodes. Each node must contain at least 2 fields: a data value field and a field containing the address of the next node in the list. For example, we can simulate a string of characters using a linked list of `alpha_node` objects declared below:

```
struct alpha_node {
   char val;
   struct alpha_node *next;
};
```

Each `alpha_node` object will contain a character value, and the address of the next `alpha_node` object in the list. We could assign and display the values of each node with the following:

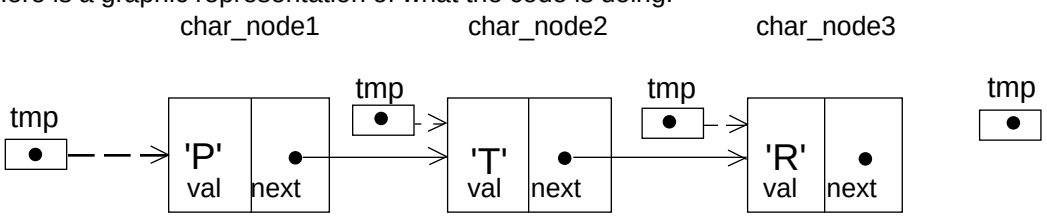
```
#include <stdlib.h>
void main ()
{
   struct alpha_node char_node1, char_node2, char_node3, *tmp;

   /* each node knows the address of the next node */
   char_node1.next = &char_node2;
   char_node2.next = &char_node3;
   char_node3.next = NULL;
   /* assign the values to each node */
   char_node1.val = 'P';
   char_node2.val = 'T';
   char_node3.val = 'R';

   /* get a starting point to traverse the list */
   tmp = &char_node1;
   /* traverse the list, printing each nodes' val field */
   while (tmp != NULL) {
      printf ("%c->",tmp->val);
      tmp = tmp->next;
   }
   printf ("END_OF_LIST_SENTINEL\n");
}
```

This produces the output: `P->T->R->END_OF_LIST_SENTINEL`

Here is a graphic representation of what the code is doing:



Solution - Linked Lists

- A linked list is a chain of data structures joined together.
- Linked lists do not usually reside in consecutive memory locations.
  - Each element in the chain knows where the next element is located.
- An element is added to or removed from the list as needed.
  - The addition or removal of an element in the list requires only a pointer adjustment.
- Linked lists are “traversed” to obtain needed information.
  - Starting with the first node, each node is “visited”, and its corresponding data value(s) reported.
- Traversal ends when a known sentinel is reached.
  - Just as the character ‘`\0`’ flags the end of a string, a pointer with value 0 is used to flag the end of a linked list.
  - The value NULL (defined in `<stdlib.h>`) represents a pointer with value 0.

Notes

Use the steps described below to append a new node onto a linked list. Create a node. Check if there are any nodes yet in the list. If not, set the `head` and `tail` pointers to the address of the new node, else if there are nodes already in the list, append the node onto the end of the existing list. Reset the `tail` pointer to the new node.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct alpha_node  {
   char val;
   struct alpha_node *next;} alpha;
void insert_node (alpha **head, alpha **tail, alpha temp);
void print_nodes (const alpha * head);
void main (void)
{
   alpha *head = NULL, *tail = NULL, tmp;
   int c;
   for (c='A'; c<'F'; c++){
      tmp.val = c;
      insert_node (&head, &tail, tmp);}
   print_nodes (head);
}
void insert_node (alpha **head, alpha **tail, alpha temp)
{
   /* initialize the new node pointer */
   alpha *newn = NULL;
   /* allocate the new node */
   if (NULL == (newn = (alpha *)malloc (sizeof (alpha)))){
      perror ("insert_node");
      exit (EXIT_FAILURE); }
   /* assign the contents to the newly allocated space */
   *newn = temp;
   /* always initialize your pointers */
   newn->next = NULL;
   /* Will this be the first node in the list? */
   if (*head == NULL){
      /* initialize the pointers */
      *head = *tail = newn; }
   else {
      /* append the new node onto the end of the list */
      (*tail)->next = newn;
      *tail = (*tail)->next; }
}
void print_nodes (const alpha *head)
{
   /* starting with the first node in the list */
   const alpha *tmp_node = head;
   /* traverse the list, printing values until you hit the NULL sentinel */
   while (tmp_node != NULL){
      printf ("%c -> ",tmp_node->val);
      /* advance the pointer to the next node in the list */
      tmp_node = tmp_node->next; }
   /* at NULL sentinel */
   printf ("NULL\n");
}
```

Linked List - Formation

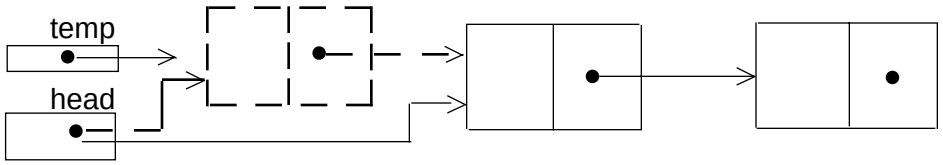
- The previous example joined three statically defined structure objects. This would rarely be the case.

- The nodes of a linked list are dynamically allocated using `malloc()` or `calloc()`.
  - You may allocate and free nodes as you need.
- We will be discussing single-linked lists.
  - Each new node is joined to a previous node, and points to (knows the location of) the next node in the list.
- The start of the list will be kept track of with a head pointer.
  - The head pointer will always contain the address of the first node in the list. If the first node should change (e.g.: it may be removed, or shifted to another position in the list), the head pointer is updated accordingly.
  - Other pointers may be defined for the list (e.g.: `previous`, `last, etc...`)
- The last node in the list will have the value `NULL` as the location of its next node.
  - `NULL` is used to flag the end of a linked list.
- There are several basic operations that are performed on linked lists:
  - Insert a new node.
  - Delete an existing node.
  - Traverse the list.
  - Search for and display a particular node in the list.
  - Display the entire list.
  - Sort the list.

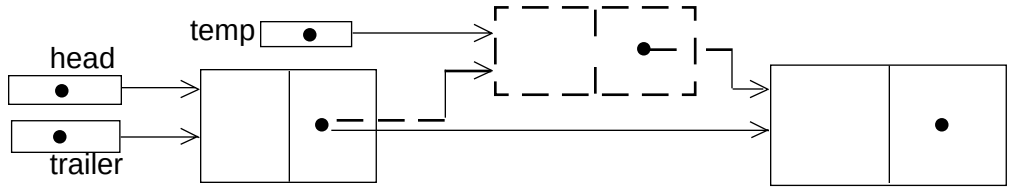
Notes

When you remove a node from the list, you must be careful to maintain your pointers. After you have located the node to be removed, there are 3 cases you should consider:

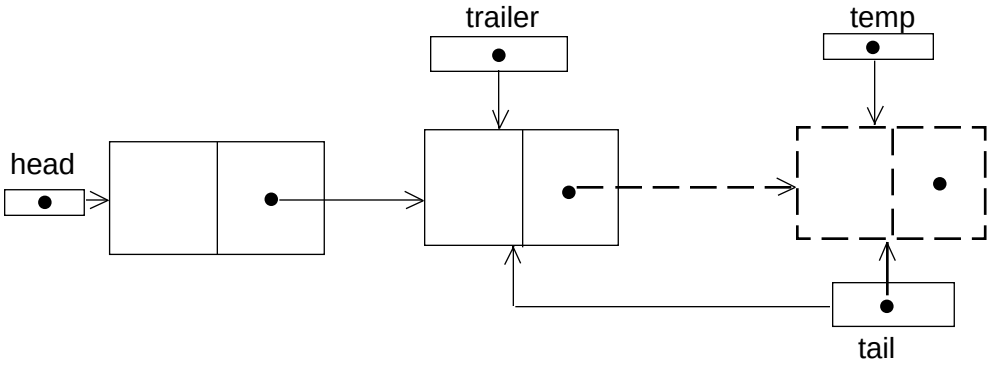
case 1) Removing the first node:
Assign a `temp` pointer to the node to be removed.
Reposition the `head` pointer to the new top of list.
Free the space referenced by the `temp` pointer.



case 2) Removing a node from the middle of a list:
Traverse the list using a `trailer` pointer to follow one node behind.
Assign a `temp` pointer to the node to be removed.
Assign the address of `temp`'s next node into `trailer`'s next field.
Free the space referenced by the `temp` pointer.



case 3) Removing the last node in the list:
Follow the steps in case 2, then check the value of the `trailer`'s next field. If it is `NULL`, then you know you're at the end of the list.
Reassign the `tail` pointer to point at the `trailer` node.



List Operations - Delete

- Removing a node requires changing the address stored in the previous node to be the location of the removed node's succesive node.
- When a node is removed from the list, return the allocated space returned back to the heap using `free()`.
  - There is only a finite amount of heap available.
  - When a node is removed from the list and not de-allocated with `free()`, there is no way of freeing the space back to the heap. This allocated space which is never freed is referred to as memory leak.

Notes

Exercises

In this set of exercises, you will be changing your existing array of structures to a linked list of structures. Change your menu options accordingly using the program developed in Chapter 4 Exercises:

1) Make any modifications to your employee structure declaration to allow for the creation of a linked list of employees.

2) Modify your function that allowed the user to insert a new employee into the array, so that it will now add employee nodes into the linked list.

3) Modify your function that allowed the user to display all employees in the array to traverse the linked list, printing all employee information.

4) Create a menu option that prompts the user for a certain employee id to display. Create a corresponding function that will locate the specified employee in the list. If found, display the employee. If not found, allow the user to enter the new employee if they choose.

5) Create a menu option that prompts the user for a certain employee id to remove from the list. Create a corresponding function that will locate the specified employee in the list. If found, remove the employee from the list. If not found, display an informative error message.

Optional:

A) Modify your insert function to insert employees ordered by their names. (Watch your pointers!)

B) Create a menu option that allows the user to delete all employees in the list.

C) At the start of the program, read existing employees from a file into a linked list. At the end of the program, write out the current list of employees to a file.

D) Provide the ability to edit any data field of an employee record.

Notes