My C Programming Language Primer

by Peter J. Philipp

Foreword

I wrote this booklet between 2008 and 2013 for people who view the online wiki http://www.hackepedia.org.

In 2017 I downloaded the source of this and reformatted it with Libreoffice. I‘d like to thank Yashy, Hawson and Xitami for helping with this creation.

In the text many hostnames were used for prompts such as mimas, dione, neptun and neptune. These were computers (also under vmware) at the time of writing this document.

This small booklet is dedicated to my parents Elisabeth and Ulrich Philipp.

November 2017
Peter J. Philipp

Table of Contents

1. Introduction....3
2. The UNIX Interface....5
3. The Internet Interface....8
4. Integers, Loops and Branches....11
5. Pointers, Variables and Variables of some sorts....15
6. Pointers and Arrays....19
7. Arguments and Environment Variables....23
8. Standard Input, Standard Output, Standard Error....25
9. Writing/Reading Files....29
10. Patching Source Code....32
11. Character Arrays and Linked Lists....33
12. Bitwise Operators....40
13. Internet programming....43
14. External Libraries....50
15. Still doesn't cut it?....52
16. Other Online Tutorials....52
17. Bibliography....52

1. Introduction

The C programming language was invented at **AT&T Bell Labs** by Dennis Ritchie.

Since **UNIX** and C are intertwined it's good to know the basics of both. The "K&R The (ANSI) C Programming Language" written by both Kernighan and Ritchie is the "bible" for C programmers. Another reference will be "The UNIX programming environment" written by Kernighan and Rob Pike. This book ties UNIX's development tools together such as simple shell programming as well as the UNIX interface of the C programming language.

The classic K&R program that the C book starts with is:

```
int
 main(void)
 {
        printf("Hello, World\n");
 }
```

You see the structure here of a program. What you see between the { and } brackets is known as a "block" in C. It ties a series of instructions together, whether in a logical branch (often an if branch) or to define a function or procedure (same thing). Every C program has a main() procedure. This is where the program starts. main() returns an integer which is signed[1]. The return value can be caught and read by the user of
the program when the program exits.

A procedure also takes one or more arguments. In the code example above the argument is void, meaning it's ignored (even if there is arguments passed to the procedure). main() usually takes 2 arguments to read arguments[2] from the program that executed the particular program. There is a third argument on UNIX systems that allows a users environment list to be passed into the program as well.

Finally the instruction, printf(). As you can see it is a function as well and it takes as the argument

"Hello, World\n".

The \n means newline (ascii hex 0a dec 10) and ensures that when the program returns to a user shell that the prompt will be on the next line. What is printed in fact is "Hello, World". Then the program exits. As there is no

return value passed with the return() or exit() function I'm not sure if it will be consistent. Consistency is key in programming.

2. The UNIX Interface

As indicated in the introduction, UNIX and C go together. The OS provides an API to hook into the kernel (privileged core of the OS) to make use of system calls. All input and output to and from a program is dealt with a system call, calculations do not and remain in userland. UNIX has two sides, a kernel and userland. Userland is for users and administrators who request services from the kernel which queues the request, and services results back as soon as possible. The kernel schedules processes (programs) in userland to run on a fair share basis (at least in theory and determined by the algorithm). Free UNIX clones or UNIX-like Operating Systems exists. These are great for personal study, and excellent for learning C because they are included with the GNU C compiler. In the old days one had to dual-boot a computer on partitions, today one can run a virtual computer in a window and run any Operating System they like. This eases switching between platforms somewhat.

If you run Mac OS X it is based on UNIX and you can open a terminal to get to the command prompt. Linux and BSD OS's also have this capability. So when you have the C source code (ending in a .c or .cc extension) how do you make it work? In C you compile the source code into a binary code which is machine language, unlike the language BASIC C is not interpreted, which means the program is a script run inside a program that knows what to do with the instructions. Compiled programs do not need another program to make themselves run. Usually a compiler (such as gcc) produces assembler text first round through, and passes that to an assembler to produce the binary code. This is a good feature and allows people familiar with assembler to really debug the execution path (order of operations) of a program. Here is a list of useful programs:

gcc, cc - C compiler (-o file outputs binary code, -static produces static code)
gdb - debugger (allows you to step through your program for every instruction)
ldd - list dynamic dependencies (if dynamically compiled, reduces size of bins)
objdump - disassembler (produces assembly language from binary code)
file - identifies a program (similar to ldd), useful!
nm - identifies symbols in the binary code of a program, probably helpful for reverse engineering although I have never done this.
vi, ed, pico - useful text editors to enter the C language code.

So the small program in the introduction can be compiled like the following:

```
cc -o helloworld helloworld.c
```

and then the binary can be executed with ./helloworld

You may need to add an ""#include <stdio.h>"" to the top, which is the standard C input/output library. The .h is a header file, but we'll get to that. (I hope). You can add -Wall to the command line arguements to show you all of the warnings. If you want to write "clean" code, you will fix it until it has no errors.

As soon as you run a program on UNIX and it ends there is an exit code. Every shell has a different way of querying the exit value as you face the prompt again, but with the /bin/sh, bash and ksh shell you can type echo $? to query the return value.

Another great feature that UNIX offers other than opening files is pipes. A pipe (symbol | on the command prompt) allows one to direct the output of one program into the input of another program and thus you create what is called a pipeline. A pipeline uses different programs that specialize on one task to shape the output at the end into something useable. Take this example of the previous text; if I did:

```
$ cat c-programming | grep -A5 ^int | cat -n
     1  int
     2  main(void)
     3  {
     4          printf("Hello, World\n");
     5  }
     6
```

The output is the source code of the helloworld.c program and the -n argument to cat adds a sequential line count per line. Ok so you can put together your own programs and programs ""already created"" for you to make the final output. There are limits, but this is almost 40 years old.

Another important point is that if you want to make custom modifications to the UNIX Operating System, you can do this and the source code is in C, minor parts are written in assembly but only on really low end stuff. The C source code guarantees that you can edit any of the code you like, or hire someone to edit it if you desire to do so. Unfortunately Apple discontinued OpenDarwin as far as I know, but there is a certain paranoia in most corporate circles that there is a loss of income if source code is revealed to scary people. It's in all our best interests to ensure this mindset is corrected by the decision makers.

Most open source operating system vendors show you the steps to turn the C

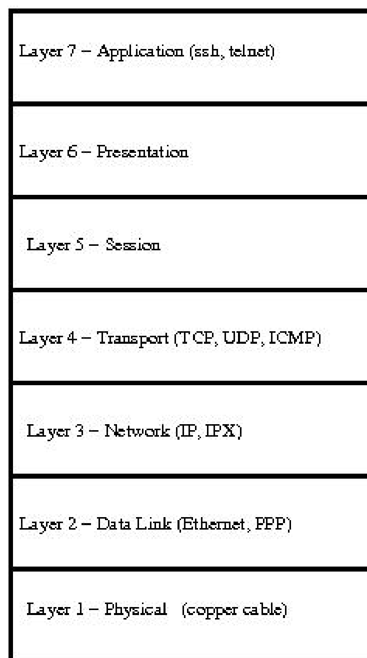
program code into binaries that run the final system. All the code has to be compiled which depending on your processor speed takes days to a few minutes and then the system requires a reboot. The reboot is required to load the new kernel which then services everything else.

Please see http://www.hackepedia.org for a lot of help on UNIX that I contributed my time to.

3. The Internet Interface

Our Internet is closely tied to UNIX and thus C. UNIX has a great set of API's for userland to connect into the greater network. The standard of Internet API's is probably Berkeley sockets which were created on the BSD variant of UNIX.

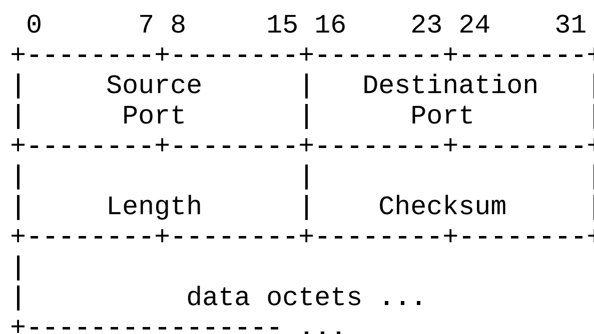
If network communication were chopped up into layers and you have physical exchange of bits over wire or radio then this would be layer 1. If you have a standard protocol with rudamentary addresses and perhaps a checksum at the back and on the Data Link you'd call this layer 2. Then if you have a more complex protocol such as the Internet Protocol that reaches beyond routers to form an internet you'd call this a Network layer or layer 3.



Finally IP carries Transport protocols such as TCP (Transmission Control Protocol) which is complex and has features such as windowing, retransmissions, sequencing, reception acknowledgement and so on. And it also carries simple protocols such as UDP (User Datagram Protocol) which really only defines a source port, a destination port, the data length and a checksum for it all. This is the Transport layer or layer 4. When dealing with TCP and UDP in UNIX the socket API allows you limited manipulation of layer 4, but most of it is done by the kernel (such as checksumming). All you're left with is sticking in the data in there and wait for the system call to return. What the system call returns determines whether you successfully sent a packet out of a computers network interface or not. You can also have RAW sockets which means you have direct control over layer 3, and you have to

write the header protocol, checksumming, payload data and other features such as retransmissions yourself. This socket mode is restricted to the superuser/root. There is also layer 2 access for the real hard-core but I won't go into that just yet.

Ok here is a diagram from RFC 768 (UDP):



User Datagram Header Format

The funny numbers at the top indicate a scale, from 0 through 31. These are bits. Each rectangle represents 16 bits in this picture and you look at it from top left to right. Something like this is easy to construct in C because it coincides with the size of short unsigned integers that are part of C. I'm going to show you an example of how this would be constructed in C, if you don't understand it, ignore it and come back to it later when you're able to.

```
/*
 * this would be a definition of a UDP header
 */

struct udpheader {
       u_int16_t sport;
       u_int16_t dport;
       u_int16_t len;
       u_int16_t csum;
};
```

Notice that a UDP packet is limited to 65535 bytes due to the integer limit that one can pack into a 16 bit integer. So if you had a 1400 byte packet you

could do the following.

```
u_char packet[1400];
 u_char *data;
 struct udpheader *uh;

 uh = (struct udpheader *)&packet[0];
 uh->sport = 1024;
 uh->dport = 53;
 uh->len = 1400;
 uh->csum = 0;

 data = &packet[sizeof(struct udpheader)];
```

There is a few caveats to this example (it's only an example). It would work on a powerpc processor, or sparc processor as written, but not on intel (i386) processors. There is a law of ordering bits of packets on the internet and they require the least significant bit to cross first right through to the most significant bit. The registers on an i386 processor store a 16 bit integer with the most significant bit first and least last. This is calledendian-ness or byte order. Network byte order is big endian as outlined, what intel processors use is little endian byte order. So you need a function to flip the bytes in the right order. UNIX has a few functions to do that. This is how you write the above portable.

```
uh->sport = htons(1024);
```

htons() - host to network (short 16 bits) swap.

Ok. The final word. As you can see with C you can write out the headers of packets easily, it gets a little tricky when things are compressed into nibbles (half-bytes - 4 bit) or even 1 bit flags. But C can work around this.

Internet, UNIX and the C programming language thus seem to be made for each other. If you like any of these you'll like the rest. I'm going to get into socket programming much later perhaps.

4. Integers, Loops and Branches

Variables in C are important. In fact they exist in all programming languages. They are used for temporary or permanent storage throughout the programs life. A processor (CPU) has a set of registers that are used to do logical operations on numbers stored in them. The storage size of these registers defines the storage size of integers available in any programming language. More on this later. Often any computer is used to do boring calculations and do these at rapid speeds over and over (loops). This is why we invented computers so that they can do these repetitive tasks at great speeds. Take a look at the following main() function:

```
1		int
2		main(void) {
3
4			int count;
5
6			count = 10;
7
8			while (count > 0) {
9				printf("Hello, World\n");
10				count = count - 1;
11			}
12		}
```

What this program does is it define a variable of type int (integer) named count (line 4). It then assigns the value 10 (decimal) to it (line 6). Then comes a while() loop. A while loop will continue to loop as long as the condition given as its argument remains true. In this case (on line 8) the condition holds true when the value of count is greater than zero. On line 9 we print our familiar string (taken from #1). Line 10 then decrements the value of count by one. This is the simplest way to decrement by one. C has a few shortcuts to decrement such as:

```
count--;
--count;
count -= 1;
```

All of these ways are synonymous (the same as) to what you see on line 10. Similarily if you wanted to increase the value of count by one you could type:

```
count = count + 1;
count++;
++count;
count += 1;
```

They all mean the same. The ++ on either side has a dual functionality which I will demonstrate here:

```
1       while (count--)
2               printf("Hello, World\n");
```

Notice a few differences. The obvious decrementor has been stuffed into the while condition and the while loop doesn't have a block bracket {}. The result will print 10 Hello Worlds like the above example. Because 10 through 1 are positive and thus logically true while will continue. As soon as count reaches the value of 0 while() will break. Consider the following difference:

```
1       while (--count)
2               printf("Hello, World\n");
```

Here our string will be printed only 9 times. The reason is the following. When a incrementor or decrementor is before a variable/integer the value is decremented before it is evaluated in a branch condition (in order to break the loop). If the decrementor is after the integer while will evaluate the value and then that value gets decreased. This allows certain freedoms in logic of code and allows a sort of compactness in order to screw with your perception and natural logic.

Much like decrementing operations C also has a few different ways to loop.

```
do {
        something;
} while(condition);

for (count = 10; count > 0; count--) {
        something;
}
```

Do/while loops are nice to haves when you have to enter a loop on a condition but have to execute the code at least once that is within the loop. This compacts having to write out the same code twice. for() loops are very popular because the three arguments (delimited (seperated) by ';'). The first argument sets a value to a variable. There can be more than one of these but they have to be delimited by a comma (,). The second argument is the check condition in order to break the loop. And the last argument is the decrementor or incrementor of a value, there can be more than one again delimited by a comma. It's a nice way to compact a loop.

I'm going to go into endless loops but before I do I'm going to introduce a simple branch in order to break out of the loop. Consider this:

```
1       while (1) {
2               printf("Hello, World\n");
3
4               if (--count == 0)
5                       break;
6       }
```

Ok line one defines the endless loop. 1 is always true and it doesn't get increased nor decreased. Line 4 introduces the if () branch, it is similar to the IF/THEN also found in BASIC. New is the == sign, and this is often in confusion. To test a value, C expects a character before an equal sign so the following combinations can work:

```
== equals
!= does not equal
<= less than or equal to
>= greater than or equal to

< less than
> greater than
```

Imagine the following scenario (typo):

```
if (--count = 0)
        break;
```

Then the loop would never exit/break because count gets decremented and then assigned the value 0 and 0 is not TRUE it is FALSE. Luckily todays GCC compiler catches this and won't compile this. The error message it spits back is:

```
test.c: In function `main':
test.c:9: error: invalid lvalue in assignment
```

Consider we wanted to skip the part where count holds the number 5, then:

```
1       while (1) {
2               if (count == 5) {
3                       count--;
4                       continue;
5               }
6
7               printf("Hello, World\n");
8
9               if (--count == 0)
10                      break;
11
12      }
```

Notice that count has to be decremented before continuing or you'd have an endless loop again. Remember all examples that have numbers to indicate their position need to have the numbers removed. Here is an example of the last program.

```
blah.c: 21 lines, 188 characters.
neptune$ cc -o blah blah.c
neptune$ ./blah | wc -l
      9
neptune$
```

The program 'wc' does a word count and when passed the argument -l it will count lines. You see here that Hello, World was printed 9 times. Thank goodness for pipes or this example would be extremely boring.

To branch on a condition you would type:

```
if (count == 0)
 printf("you have none left!\n");
else if (count == 1)
 printf("you have one left!\n");
else
 printf("there is plenty left...\n");
```

This is how you would fulfil several branches.

Also pretend you use switch()...

```
switch (count) {
case 0:
  printf("you have none left!\n");
  break;
 case 1:
  printf("you have one left!\n");
  break;
 default:
   printf("you have plenty left...\n");
   break;
 }
```

The break; here is optional:

```
switch (count) {
case 0:
 printf("you have none left!\n");
 break;
 case 1:
   /* FALLTHROUGH */
 default:
 printf("still some left...\n");
```

I hope this explains some more branches.

5. Pointers, Variables and Variables of some sorts

Pointers are often seen as something hard to understand, and it is true that often C programs have their bugs near pointers. I think the problem is psychological when people think there is difficulty. I'm going to start fairly early with pointers so that they are covered right away.

In C there is different types of variables. One we covered with loops already and that is the Integer (int). There is a few others.

```
1. short
2. int
3. long
4. long long
5. char
6. void
7. struct
8. union
```

1. short

short is a short Integer of size 16 bits (2 bytes). If it's signed it can hold 32767 as maximum, after that it wraps around into the negative. More on this later. Unsigned limit is 65535.

2. int

Integer. We shortly covered this in the last article. An integer is looked at as a 32 bit entity (4 bytes). Signed limit is 0x7fffffff (hexadecimal) and unsigned limit is 0xffffffff. Please refer to the /usr/include/limits.h header file and follow all inclusions. There is aliases for most limits.

3. long

A long integer. On 32 bit architectures this is 32 bits (4 bytes), and on 64 bit architectures this should be 64 bits (8 bytes). One should put this into consideration when writing portable code for different architectures.

A new way of defining integers is to write out what they are in the code, these are aliases to the defined types. You have the following:

```
uint8_t, int8_t - 8 bit unsigned and signed integer
uint16_t, int16_t - 16 bit unsigned and signed integer
uint32_t, int32_t - 32 bit unsigned and signed integer
uint64_t, int64_t - 64 bit unsigned and signed integer
```

Now you take away the confusion but must write your own aliases (#define's)

for them if you want to compile these integers on old computers with old compilers. Do understand that using a 64 bit integer on a 32 bit architecture is most likely going to result in having to split the integer over 2 registers, this is a performance degradation and possibly not what you want when you count on speed.

4. long long

I believe what is meant here is the same as a int64_t (8 bytes).

5. char

A char holds a byte (8 bit). It is synonymous to 8int_t. u_char and char both take up 8 bits and can be used interchangably.

6. void

This is a stub. It is used to indicate nothing you can often see this in C source code when a return value of some system call is being ignored such as:

```
(void)chdir("/");
```

The brackets indicate that it is "casted". The system call chdir("/"); on unix systems should always work as UNIX cannot run without a root filesystem so you don't need to check for an error condition, thus void. void consumes 32 bits I believe.

7. struct

struct is a special type. It comprises an object comprised out of 1 or more other variables/objects. You can build IP packet headers with structs as shown in the previous example or just have a grouping of 2 integers. Here is an example:

```
struct somestruct {
        int a;
        int b;
 } ST;
```

Accessing integer a, b then you could use:

```
  ST.a = 7;
  ST.b = 12;

  if (ST.a > ST.b)
        exit(1);
```

[These are just examples and don't say anything in case you're trying to read into these.]

Alternatively the above struct can be defined like so:

```
 struct somestruct {
        int a;
        int b;
 };

struct somestruct ST;
```

Both ways have the same results.

8. union

A union is built up like a struct but the individual members overlap on each other. This is great when you want to exchange values between different sized variables/objects. Consider this:

```
 union someunion {
        char array[4];
        int value;
 } US;
```

You can then fill the value in the union US and read the individual bytes of that value from a character array of the same length. I'll get to arrays a little further down. Pretend you want to change an IP (version 4 - at the time of this writing the current) address represented as a 32 bit number and write out the dotted quads (as they are called in the networking world) then you'd have something like.

```
        US.value = somevalue;
        printf("%u.%u.%u.%u\n", US.array[0], US.array[1], US.array[2],
US.array[3]);
```

The example is written for a big-endian machine, in order to make it portable with little endian (little byte order) machines such as intel or amd processors. You need to change it given the htonl(), ntohl() functions. Read the manual pages found online on UNIX systems for these functions.

Another good way for a union is to find the byte order of a machine in the first place. Pretend somevalue is 0x01020304 (hexadecimal) then on big endian machines you'd see 1.2.3.4 and on little endian machines you should see 4.3.2.1. The order is reversed where MSB (most significant byte) becomes LSB (least significant byte).

An example on an amd64 computer:

```
neptune$ cc -o testp test.c
neptune$ ./testp
4.3.2.1
```

I don't have a G3 or G4 Macintosh handy at the moment but you'd most likely see 1.2.3.4 on that computer.

6. Pointers and Arrays

Every variable type is represented by an address in the memory of your computer. That address stores the value of that variable. Pointers allow you to store another address. Pretend you have a 32 bit computer and it is capable of manipulating address space starting at address 0 all the way up to 0xffffffff (hexadecimal). This gives you a limit of 4 gigabytes of memory. So when you want to read memory in C you can use pointers. Take this example:

```
int value = 1;
int anothervalue = 2;
int *pv = NULL;
```

Notice the asterisk (star) before pv. This is a pointer to an integer and it is declared to point to NULL (a macro representing 0).

```
pv = &value;
printf("%d\n", *pv);
printf("%u\n", pv);
pv = &anothervalue;
printf("%d\n", *pv);
printf("%u\n", pv);
```

Consider this. when you assign the address of "value" to pv you can then print the value of "value" by adding an asterisk in front of pv. To print the address that "value" resides in memory you'd just print pv. In order to print the address of any value you prepend it with an ampersand (&). It's straight forward. Watch how this program executes on an amd64 64 bit system. Here's the program first, notice I changed the variables from int to long in order to fit all 64 bits of address space.

```
1       #include <stdio.h>
2
3       int
4       main(void)
5       {
6               long value = 1;
7               long anothervalue = 2;
8               long *pv = NULL;
9
10              pv = &value;
11              printf("%ld\n", *pv);
12              printf("%lu\n", pv);
13              pv = &anothervalue;
14              printf("%ld\n", *pv);
15              printf("%lu\n", pv);
16      }
```

```
neptune$ cc -o testp test.c
neptune$ ./testp
```

```
1
140187732443816
2
140187732443808
```

Notice the output. Those addresses are really high numbers telling you a few things. Since addressed memory starts at 0 and grows to a maximum of 0xffffffffffffffff (hexadecimal) on 64 bit computers there is a lot of expansion room for RAM. The computer I use has 1 gigabyte of memory. But if you look at the address of value and another value it goes way beyond 1 billion. There must be memory holes between 0 and that number. And this is true because of memory address translation (MAT) which is used in UNIX.
The physical memory addresses are translated to virtual memory in order to protect other memory of other users in the system. This is all handled by the kernel (OS) and is invisible to the user and often irrelevant to the C programmer. Another interesting thing you'll notice is that the two addresses are 8 bytes of address space apart. Exactly the storage size of a long integer (64 bits). On a 32 bit system (i386) this would be 4 bytes.

So pointers point to an address in memory, and because they have a type they can also manipulate bytes starting at that address (offset). This is a useful feature.

On to arrays. You've already seen a character array, and I'll continue on this thread a little bit. Consider this program:

```
1       int
2       main(void)
3       {
4               char array[16];         /* array[0] through array[15] */
5               int i;
6
7               for (i = 0; i < 16; i++) {
8                       array[i] = 'A';
9               }
10
11              array[15] = '\0';
12
13              printf("%s\n", array);
14      }
15
```

Notice the for() loop rushing from 0 through 15, at value 16 it's not less than 16 anymore and thus the loop breaks. An array in C always starts at 0 upwards. This is confusing at first but you get used to it (you also start at address 0 in the computers memory and not 1). On line 11 the 16th character is replaced with a NULL terminator which is equivalent to '\0'. Finally on line 13 the array is printed, here is the output:

```
test.c: 17 lines, 195 characters.
```

```
neptune$ cc -o testp test.c
neptune$ ./testp
AAAAAAAAAAAAAAA
neptune$ ./testp | wc -c
      16
```

Notice wc -c returns 16 because it counts the newline '\n' as well. In C every string has to be terminated with NULL, or it cannot be printed with the %s argument to printf(). Consider this small modification with pointers:

```
1       #include <stdio.h>
2
3       int
4       main(void)
5       {
6               char array[16];         /* array[0] through array[15] */
7               char *p;
8               int i;
9
10              for (i = 0; i < 16; i++) {
11                      array[i] = 'A';
12              }
13
14              array[15] = '\0';
15
16              p = &array[0];
17
18              while (*p) {
19                      printf("%c", *p++);
20              }
21
22              printf("\n");
23      }
24

test.c: 24 lines, 254 characters.
neptune$ cc -o testp test.c
neptune$ ./testp
AAAAAAAAAAAAAAA
```

Same output as before but this time what was printed was done so one character at a time. The p pointer is assigned to point to the beginning of array. We know it's a string (because of the termination) so we can traverse through the array in a loop printing the value that p points to (asterisk *) and then incrementing the value of the pointer address by one. Eventually *p on line 18 will return the NULL and to while() this is FALSE and the loop will break. Then we print a newline in order to make it pretty for the next prompt. So why doesn't the value of *p increase with ++? You'd put brackets around it like so:

```
printf("%c", (*p)++);
p++;
```

But that won't do much because the increment is after the value has been

passed to printf(). This would be better:

```
printf("%c", ++(*p));
p++;

test.c: 25 lines, 263 characters.
neptune$ cc -o testp test.c
neptune$ ./testp
BBBBBBBBBBBBBBB
```

That's how it would look like.

You can replace the char type on any of these with short or int types it doesn't matter the concept is the same. Obviously you won't be able to print these types but you can work on loops that count the size of the
array. The example with the while() loop for the pointers only works on NULL terminated strings (character arrays).

7. Arguments and Environment Variables

A program by definition takes input, processes the input and spit back output. In UNIX a lot of input is given to programs from the shell command line. Consider this following shell program:

```
mimas$ cat shellprogram.sh
#!/bin/sh
echo $# arguments
while [ ! -z $1 ]; do
 echo $1
 shift
done
mimas$
```

Running this program may look like this:

```
mimas$ chmod +x shellprogram.sh
mimas$ ./shellprogram.sh argument1 argument2
2 arguments
argument1
argument2
mimas$
```

As you can see it counts the arguments lists them and then prints the arguments given. We want to write this in C. And I'm telling you right now that you'll get a better understanding of UNIX and C when you've done this following program. To start off let's take a look what happens when we execute a program. The shell commandline starts a new program by calling fork() and then exec() which has the option of passing the arguments and environment variables. The program that gets executed then takes that information through special variables. Consider this program in C:

```
mimas$ cat -n program.c
     1  #include <stdio.h>
     2
     3  int
     4  main(int argc, char *argv[], char *environ[])
     5  {
     6          int i;
     7
     8          printf("%d arguments\n", argc - 1);
     9
    10          for (i = 1; i < argc; i++)
    11                  printf("%s\n", argv[i]);
    12
    13          return (0);
    14  }
```

(the line numbers are produced by cat -n output, they aren't part of the program). As you can see from examples above the line main(void) changed to main(int argc, char *argv[], char *environ[]). This is what exec passes to

the newly started program. ARGC (lower caps) is the "argument count", an integer, and ARGV(lower caps) is the argument value which is an array of type char *, next is ENVIRON(lower caps) which is the environment as an array of char *. Compiling and executing this program looks like this:

```
mimas$ cc -o program program.c
mimas$ ./program argument1 argument2
2 arguments
argument1
argument2
mimas$
```

The output looks exactly like the shell script we wrote above. This is then how you manipulate the input data through the argument variables. Do note that the "char *environ[]" can be left out if environment variables aren't used but if they are and you want to see how to use them then read on:

Consider this change in the program:

```
mimas$ cat -n program.c
     1  #include <stdio.h>
     2
     3  int
     4  main(int argc, char *argv[], char *environ[])
     5  {
     6          char **p = environ;
     7
     8          while (*p) {
     9                  printf("%s\n", *p);
    10                  p++;
    11          }
    12
    13          return (0);
    14  }
mimas$
```

The pointer to an array of char's is defined on line 6. A while loop on line 8 tests whether *p is NULL or not. And if it's not NULL (which is the terminator, end of array) then print out what *p points to. Then increase p to the next element on line 10.

Your environment is likely long and mine is too, so I'm going to pipe the output into wc -l to just show you a line count of the output:

```
mimas$ ./program | wc -l
      24
mimas$
```

As you can see I had 24 environment variables that wc counted up.

Now you're able to give your programs input such as the "-l" in "wc -l". How wc does its input from the other program may be written in the next chapter.

8. Standard Input, Standard Output, Standard Error

Last chapter we learned that a program can take input from arguments and environment variables. In this chapter we're going to read from standard input. In UNIX (unless a daemon) a program always has 3 filedescriptors open called standard input, standard output and standard error. These 3 standard methods make pipelines possible and we'll explore how wc works. Consider the below program:

```
 1  #include <sys/types.h>
 2  #include <unistd.h>
 3  #include <stdio.h>
 4
 5  int
 6  main(int argc, char *argv[])
 7  {
 8          int wcount = 0;
 9          int lcount = 0;
10          char c;
11
12          while (read(STDIN_FILENO, (char *)&c, 1) > 0) {
13                  if (c == '\n')
14                          lcount++;
15
16                  wcount++;
17          }
18
19          printf("words: %d\n", wcount);
20          printf("lines: %d\n", lcount);
21
22          return (0);
23  }
```

We introduce a system call on line 12 called read. In UNIX most input/output is managed by system calls so I'm going to introduce you to read() instead of getc() which is a wrapper to read(). A wrapper is a program that nicens the interface to a sytem call or other function. Anyhow, read returns 0 on EOF (end of file) and thus the while loop breaks when we have a return value of 0 or lower. This is what it looks like when we run it:

```
mimas$ cc -o wc-clone wc-clone.c
mimas$ echo hi| ./wc-clone
words: 3
lines: 1
```

The program is somewhat incorrect in that it calls a character a word (sorry). But it prints the count of the three characters 'h', 'i' and newline. It counts the newline which are represented on line 13 as '\n'.

```
 5  int
 6  main(int argc, char *argv[])
```

```
     7  {
     8          int ccount = 0;
     9          int wcount = 0;
    10          int lcount = 0;
    11          char c;
    12  
    13          while (read(STDIN_FILENO, (char *)&c, 1) > 0) {
    14                  if (c == '\n') {
    15                          lcount++;
    16                          wcount++;
    17                  } else if (c == ' ')
    18                          wcount++;
    19                  else if (c == '\t')
    20                          wcount++;
    21  
    22                  ccount++;
    23          }
    24  
    25          printf("\t%d\t%d\t%d\n", lcount, wcount, ccount);
    26          return (0);
    27  }
```

Here the program has been changed a little to look more like the real wc, we also count up whitespaces (\n, \t, ' ') for a word count and have named the character count right. Check out the differences.

```
mimas$ cc -o wc-clone wc-clone.c
mimas$ cat /etc/passwd | wc
     57     125    3125
mimas$ cat /etc/passwd | ./wc-clone
        57      125     3125
mimas$
```

Looks like the same output to me so our program is on the right track. Now we want to add the flags like -l in wc -l.

```
mimas$ cat -n wc-clone.c
     1  #include <sys/types.h>
     2  #include <unistd.h>
     3  #include <stdio.h>
     4  
     5  int
     6  main(int argc, char *argv[])
     7  {
     8          int i, lflag = 0;
     9          int ccount = 0, wcount = 0, lcount = 0;
    10          char c, ch;
    11  
    12          while ((ch = getopt(argc, argv, "l")) != -1) {
    13                  switch (ch) {
    14                  case 'l':
    15                          lflag = 1;
    16                          break;
    17                  default:
    18                          fprintf(stderr, "usage: wc [-l]\n");
    19                          exit(1);
    20                  }
    21          }
    22  
```

```
23          while (read(STDIN_FILENO, (char *)&c, 1) > 0) {
24                  if (c == '\n') {
25                          lcount++;
26                          wcount++;
27                  } else if (c == ' ')
28                          wcount++;
29                  else if (c == '\t')
30                          wcount++;
31
32                  ccount++;
33          }
34
35          if (lflag)
36                  printf("%d\n", lcount);
37          else
38                  printf("\t%d\t%d\t%d\n", lcount, wcount, ccount);
39
40          return (0);
41  }
```

Now the program gets a little more complex, let's look at it. On line 12 we added a getopt() function. This is a wrapper to parsing flags/arguments and is pretty handy. It only parses the the -l flag and sets it if -l is contained on the argument list (line 15). If a different flag is passed to it it will print an error message on stderr and exit (line 19).

Line 35 tests if we specified an l-flag and prints only the lcount, otherwise it will print the regular triplet values linecount, wordcount, character count.

Let's see how it works..

```
mimas$ echo hi | ./wc-clone -i
wc-clone: unknown option -- i
usage: wc [-l]
mimas$ echo hi | ./wc-clone -l
1
mimas$ echo hi | ./wc-clone
        1       1       3
mimas$
```

You may have noticed that the above program has a bug. If you spotted it, congratulations! Otherwise let me show you what's wrong with it.

```
mimas$ echo "   " | ./wc-clone
       1       3       4
mimas$ echo "   " | wc
      1       0       4
```

Notice the word count being erroneous. The fix is setting a variable when the previous character was a white space...

```
...
 9          int lastwhite = 1;
...
24          while (read(STDIN_FILENO, (char *)&c, 1) > 0) {
```

```
25                  if (c == '\n') {
26                          lcount++;
27                          if (lastwhite != 1) {
28                                  wcount++;
29                                  lastwhite = 1;
30                          }
31                  } else if (c == ' ') {
32                          if (lastwhite != 1) {
33                                  wcount++;
34                                  lastwhite = 1;
35                          }
36                  } else if (c == '\t') {
37                          if (lastwhite != 1) {
38                                  wcount++;
39                                  lastwhite = 1;
40                          }
41                  } else
42                          lastwhite = 0;
43
44                  ccount++;
45          }
...
```

Something like this. Sorry for the partial program. So when we run it it looks something like this now:

```
mimas$ echo "   " | ./wc-clone
       1       0       4
mimas$ echo "   " | wc
      1       0       4
```

The code looks a little off, consider this "improvement" (same functionality)

```
24          while (read(STDIN_FILENO, (char *)&c, 1) > 0) {
25                  switch (c) {
26                  case '\n':
27                          lcount++;
28
29                          /* FALLTHROUGH */
30                  case ' ':
31                  case '\t':
32
33                          if (lastwhite != 1) {
34                                  wcount++;
35                                  lastwhite = 1;
36                          }
37                          break;
38                  default:
39                          lastwhite = 0;
40                          break;
41                  }
42
43                  ccount++;
44          }
```

It looks a bit nicer than the "nested" if's that repeat themselves.

9. Writing/Reading Files

In order for a file to be read or written to it must be opened. The stdio.h library provides the fopen() function to open a file, which provides a wrapper function to the open system call. We'll deal with bot here. Remember last chapter we said a wrapper function is a function that "wraps" a system call with another API (or something like that).

When a file is opened for reading usually its contents are read in a loop, let me show you a program that opens a file for reading and prints its contents out to stdout.

```
 1  #include <stdio.h>
 2  
 3  int
 4  main(int argc, char *argv[])
 5  {
 6          FILE *fp;
 7          char buf[512];
 8  
 9          if (argc != 2) {
10                  fprintf(stderr, "usage: filetest file\n");
11                  exit(1);
12          }
13  
14          fp = fopen(argv[1], "r");
15          if (fp == NULL) {
16                  perror("fopen");
17                  exit(1);
18          }
19  
20          while (fgets(buf, sizeof(buf), fp) != NULL) {
21                  printf("%s", buf);
22          }
23  
24          fclose(fp);
25          exit(0);
26  }
```

On line 9 we check that there is exactly 2 arguments, the name of the program and the name of a file. On line 14 we use fopen to open the filename provided by the argument #1. "r" means read mode, "w" would mean write mode but we'll get to that later. On line 15 the FILE pointer fp is checked against NULL, fopen will return NULL when there is an error. On line 16 perror prints the error and line 17 exits the program with an exit value of 1 to indicate that there was an error. Line 20 begins a while() loop testing the return value of fgets() and if it's NULL which indicates an error or end of file. The buffer buf is filled up to 512 characters and then printed on line 21 repeatedly considering there is more lines. fgets() is line-buffered meaning

that it returns upon hitting a '\n' character. Finally line 24 closes the file pointer with fclose() and we return with an exit value of 0 on line 25.

```
mimas$ cc -o filetest filetest.c
mimas$ ./filetest /var/run/syslog.pid
9864
mimas$
```

The above prints the contents of the file /var/run/syslog.pid. Believe it or not we've recreated a simple version of the program "cat".

Consider the following small program using the system calls open, read and write:

```
 1  #include <sys/types.h>
 2  #include <unistd.h>
 3  #include <fcntl.h>
 4  #include <stdio.h>
 5  #include <stdlib.h>
 6  #include <string.h>
 7
 8  int
 9  main(int argc, char *argv[])
10  {
11          int in, out;
12          int len;
13          char buf[512];
14
15          if (argc != 3) {
16                  fprintf(stderr, "usage: copy infile outfile\n");
17                  exit(1);
18          }
19
20          if (((in = open(argv[1], O_RDONLY, 0)) < 0) ||
21           ((out = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0644)) < 0) ) {
22                  perror("open");
23                  exit(1);
24          }
25
26          while ((len = read(in, buf, sizeof(buf))) > 0) {
27                  if (write(out, buf, len) < 0) {
28                          perror("write");
29                          exit(1);
30                  }
31          }
32
33          if (len < 0) {
34                  perror("read");
35                  exit(1);
36          }
37
38          close(in);
39          close(out);
40
41          exit(0);
42  }
```

Line 20 is a logical OR operation and will perror("open") if any of the opens return with -1. On line 21 open takes the arguments O_CREAT, O_WRONLY, and O_TRUNC, these are necessary to create a file if it does not exist yet, write to that file and truncate it first if it does exist. On line 26 read returns 0

at EOF and -1 on error. We loop through write for as long as there is data to read. If you've read the usage: you know this is a simple cp (we call it copy).

So that's how you use open() vs. fopen(), system call vs. wrapper function.

10. Patching Source Code

In the professional coding circles "patches" make the rounds. These aren't band-aid patches like you may imagine but rather strips of source code that either get added or taken away from the main source. In the UNIX world the program diff is used and many have standardised on the unified diff format. Let me give you an example. The last chapters program has an include statement that doesn't need to be in there, but rewriting the whole thing wastes a lot of space, so here is how I create the patch:

```
neptun:~$ cp file.c file.c.orig
neptun:~$ grep -v string.h file.c.orig > file.c
neptun:~$ diff -u file.c.orig file.c
--- file.c.orig  2010-09-12 16:39:43.794140072 +0200
+++ file.c 2010-09-12 16:40:00.104140045 +0200
@@ -3,7 +3,6 @@
 #include <fcntl.h>
 #include <stdio.h>
 #include <stdlib.h>
- #include <string.h>

 int
 main(int argc, char *argv[])
neptun:~$
```

diff -u means the unified diff method and the original file (or old file) is the first argument and then the file that's changed. Usually this result is stored in a file called file.patch and the program patch patches it into what it should look like. Watch:

```
neptun:~$ diff -u file.c.orig file.c > file.patch
neptun:~$ cp file.c.orig file.c
neptun:~$ grep string.h file.c
 #include <string.h>
neptun:~$ patch -p0 < file.patch
patching file file.c
neptun:~$ grep string.h file.c
neptun:~$
```

So this is how you use diff and patch. Pros that use CVS use cvs diff instead of just diff but the concept is the same.

11. Character Arrays and Linked Lists

If you've followed the 10 previous chapters you may have learned quite a bit. We know now how to create a program that perhaps reads from a configuration file and sets variables based on that configuration file. This is simple. But what if the configuration file holds lists of input such as IP numbers? Pretend you're writing a program that pings IP numbers every minute and does some action if an IP does not ping. You could re-read the IP list every loop but this is likely going to be slow and the costs are much higher due to access to the disk device every time. What you could do is buffer (or cache) the list of IP's in memory and read from that instead, saving you the disk access to only 1 time. Consider this next lengthy program:

```
  1      #include <stdio.h>
  2      #include <stdlib.h>
  3      #include <string.h>
  4      #include <unistd.h>
  5
  6      #define MAX_IP 100
  7
  8      char ** read_config(char *configfile);
  9      void pinger(char *address);
 10
 11      int
 12      main(int argc, char *argv[])
 13      {
 14            char *configfile = "configfile";
 15            char **list, **plist;
 16
 17            int ch;
 18
 19            while ((ch = getopt(argc, argv, "f:")) != -1) {
 20                  switch (ch) {
 21                  case 'f':
 22                        configfile = optarg;
 23                        break;
 24                  default:
 25                        fprintf(stderr, "usage: pingd [-f configfile]\n");
 26                        exit(1);
 27                  }
 28            }
 29
 30            if ((list = read_config(configfile)) == NULL)
 31                  exit(1);
 32
 33            printf("press control-c to exit this program\n");
 34            for (;;) {
 35                  plist = list;
 36                  while (*plist != NULL) {
 37                        pinger(*plist);
 38                        plist++;
 39                        sleep(1);
 40                  }
 41
```

```
42                      sleep(60);
43              }
44      }
45
46      char **
47      read_config(char *configfile)
48      {
49              FILE *f;
50
51              static char *list[MAX_IP];
52              char buf[512];
53
54              int i = 0, len;
55
56              if ((f = fopen(configfile, "r")) == NULL) {
57                      perror("fopen");
58                      return (NULL);
59              }
60
61              while (fgets(buf, sizeof(buf), f) != NULL) {
62                      if (buf[0] == '#')
63                              continue;
64                      len = strlen(buf);
65                      if (buf[len - 1] == '\n')
66                              buf[len - 1] = '\0';
67
68                      if (i > 100) {
69                              fprintf(stderr, "can only have %d IP's to ping\n",
MAX_IP);
70                              return (NULL);
71                      }
72
73                      if ((list[i++] = strdup(buf)) == NULL) {
74                              perror("strdup");
75                              return (NULL);
76                      }
77              }
78
79              list[i] = NULL;
80              fclose(f);
81
82              return (list);
83      }
84
85      void
86      pinger(char *address)
87      {
88              printf("pinging %s\n", address);
89      }
```

* Lines 8-9 are the declaration of the functions read_config and pinger.
* Line 14 declares a string called configfile that holds "configfile"
* Line 19-27 are the getopt() argument processing routines that allow an alternative configfile (-f) to be set.
* Line 30 calls the read_config function.
* Lines 34-43 are an endless loop that calls pinger with the addresses contained in char **list, then sleeps 60 seconds before repeating.

* Lines 47 is the definition of function read_config.
* Line 49 is a FILE handle called f.
* Line 56 opens the configfile with fopen() in readonly mode.

* Line 61 reads the file configfile with fgets()
* Line 62 skips any line that has a # at the first character (this could be a comment).
* Line 64-66 strips any newline character from the read line.
* Line 68 double checks that we don't go beyond our allocated storage size of pointers in list.
* Line 73 strdup()'s the buf, this gets the length with strlen(), mallocs enough space and copies what buf is holding,
the address that's returned is set as a pointer into list[]'s i'th element.
* Line 79 NULL terminates the list.
* Line 82 returns the list or rather the pointer to the lists address.

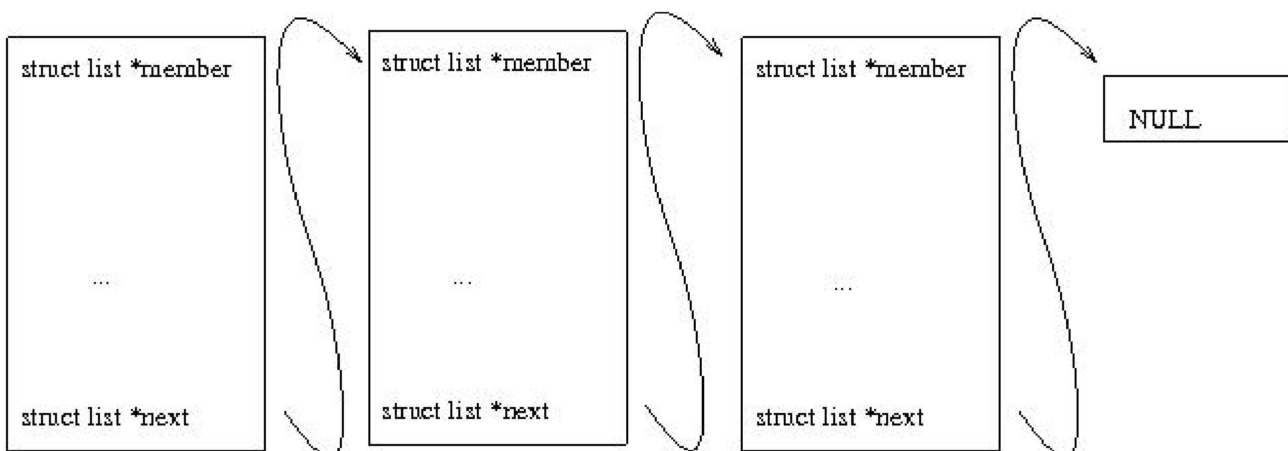
* Line 85 through 89 are the definition of the pinger() function, we may visit this again one day.

A linked list consists of a struct that links itself with another struct of the same type. Watch.

```
struct remote {
  in_addr_t ia;
  int connects;
  char *address;
  struct remote *next;
} remote_connections;
```

Pretend the above is a structure that keeps track of remote connections to your computer, it has an integer to store the IP address (ia), the number of connects (connects) as an integer and a human readable address pointer called address. It also has itself as member *next. This next pointer is NULL when there is no more links. When the linked list is created there may be several links in the "chain" and the last one is terminated with NULL. Now a program can "walk" this list by starting at the first link and then going to the *next link.

Here is a picture that may help you picture a linked list:




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

```
2       #include <stdlib.h>
3       #include <string.h>
4
5       struct list {
6             char *address;
7             struct list *next;
8       } slist;
9
10      void
11      init_list(struct list *list)
12      {
13            list->address = NULL;
14            list->next = NULL;
15      }
16
17      void
18      add_list(char *address, struct list *list)
19      {
20            struct list *new, *sl = list;
21
22            while (sl->next != NULL)
23                  sl = sl->next;
24
25            new = malloc(sizeof(struct list));
26            if (new == NULL) {
27                  perror("malloc");
28                  exit(1);
29            }
30
31            new->next = NULL;
32            new->address = strdup(address);
33
34            sl->next = new;
35            return;
36      }
37
38      void
39      walk_list(struct list *list)
40      {
41            struct list *sl = list;
42            int i = 1;
43
44            for (sl = sl->next; sl != NULL; sl = sl->next) {
45                  printf("link #%u has address %s\n", i++, sl->address);
46            }
47
48            return;
49      }
50
51
52      int
53      main(int argc, char *argv[])
54      {
55            char **p;
56
57            init_list(&slist);
58
59            p = argv;
60            for (p++; *p != NULL; p++) {
61                  add_list(*p, &slist);
62            }
```

```
63
64            walk_list(&slist);
65
66            return (0);
67      }
```

The execution of the program now looks like this:

```
neptun:~$ cc -o slist slist.c
neptun:~$ ./slist 10.0.0.1 10.0.0.2 10.0.0.3 10.0.0.4
link #1 has address 10.0.0.1
link #2 has address 10.0.0.2
link #3 has address 10.0.0.3
link #4 has address 10.0.0.4
```

* on line 5 through 8 the struct list is defined with an instance of slist
* line 10 is function init_list() which basically puts the values of NULL on address and next members.
* line 17 is function add_list() which takes input as a string and a struct list of the first member of the linked list.
* line 38 is function walk_list() which allows one to walk the linked list given as an argument and prints what address is contained in the struct member address.
* and line 52 is function main() which calls these functions, the process is all pretty straight forward if you understand while and for loops.

Interesting is following this through a debugger (watch):

```
neptun:~$ cc -g -o slist slist.c
neptun:~$ gdb ./slist
...
(gdb) break walk_list
Breakpoint 1 at 0x40070d: file slist.c, line 41.
(gdb) run 10.0.0.1 10.0.0.2
Starting program: /home/xxx/slist 10.0.0.1 10.0.0.2

Breakpoint 1, walk_list (list=0x601050) at slist.c:41
41          struct list *sl = list;
(gdb) n
42          int i = 1;
(gdb)
44          for (sl = sl->next; sl != NULL; sl = sl->next) {
(gdb)
45                printf("link #%u has address %s\n", i++, sl->address);
(gdb)
link #1 has address 10.0.0.1
44          for (sl = sl->next; sl != NULL; sl = sl->next) {
(gdb) n
45                printf("link #%u has address %s\n", i++, sl->address);
(gdb)
link #2 has address 10.0.0.2
44          for (sl = sl->next; sl != NULL; sl = sl->next) {
(gdb) print sl->next
$1 = (struct list *) 0x0
(gdb) print *sl
$2 = {address = 0x602070 "10.0.0.2", next = 0x0}
(gdb) n
49    }
(gdb) n
main (argc=3, argv=0x7fffffffe308) at slist.c:66
66          return (0);
```

```
(gdb) cont
Continuing.

Program exited normally.
```

As you can see the last link has the address 10.0.0.2 and next is NULL (0x0).

The difference between the two programs that are listed in this chapter is that one is limited to IP_MAX (in our example it's hardcoded to 100) and the second has theoretically no limit, with the exception of system memory and limit of arguments (which could be bypassed if a configfile were in place).

Consider now a "doubly linked list" which has a previous and a next link. Here is the struct for that:

```
struct list {
 struct list *prev;
 struct list *next;
 char *address;
};
```

Now the first link created has the prev set to NULL and the last like before also set to NULL. With a doubly linked list walking backwards through the list is possible, consider this example:

```
    41      void
    42      walk_list(struct list *list)
    43      {
    44              struct list *sl = list;
    45              int i = 1;
    46
    47              while (sl->next != NULL)
    48                      sl = sl->next;
    49
    50              for (; sl->prev != NULL; sl = sl->prev) {
    51                      printf("link #%u has address %s\n", i++, sl->address);
    52              }
    53
    54              return;
    55      }
```

As you can see I only showed walk_list() here line 47 places sl to the last link in the doubly linked list, line 50 then walks backwards through via the prev links printing the addresses as we go, the order of the addresses is now reversed.

```
neptun:~$ ./dlist 10.0.0.1 10.0.0.2 10.0.0.3
link #1 has address 10.0.0.3
link #2 has address 10.0.0.2
link #3 has address 10.0.0.1
```

See if you can duplicate the program dlist.c you have function walk_list and only need to modify function add_list() and init_list() minorly.

In BSD and Linux there is a set of macros that allow linked lists to be managed, consider this program that uses the queue(3) functions:

```
 1      #include <sys/types.h>
 2      #include <sys/queue.h>
 3      #include <unistd.h>
 4      #include <stdio.h>
 5      #include <stdlib.h>
 6      #include <string.h>
 7
 8      SLIST_HEAD(listhead, list) head;
 9
10      struct list {
11            SLIST_ENTRY(list) entries;
12            char *address;
13      } *l1, *l2, *lp;
14
15      int
16      main(int argc, char *argv[])
17      {
18            char **p;
19            int i;
20
21            SLIST_INIT(&head);
22
23            for (p = argv, p++; *p != NULL; p++) {
24                  l1 = malloc(sizeof(struct list));
25                  l1->address = strdup(*p);
26
27                  SLIST_INSERT_HEAD(&head, l1, entries);
28            }
29
30            i = 1;
31            SLIST_FOREACH(lp, &head, entries) {
32                  printf("link #%d has address %s\n", i++, lp->address);
33            }
34
35            return (0);
36      }
```

The macros are simpler and are used in kernel programming and some userland programming, it's good to get to know them instead of home-rolling (as it's called) linked lists. Study the man pages on queue(3) by typing "man queue".

12. Bitwise Operators

Consider this grep from /usr/src/sys/net/if_ethersubr.c:

```
mimas$ grep -e '&=' -e '|=' -e '>>' -e '<<' if_ethersubr.c
                        m->m_flags |= M_BCAST;
                        m->m_flags |= M_MCAST;
        if (!(netisr & (1 << NETISR_RND_DONE))) {
                atomic_setbits_int(&netisr, (1 << NETISR_RND_DONE));
                        m->m_flags &= ~M_PROTO1;
                        m->m_flags &= ~(M_BCAST|M_MCAST);
                *cp++ = digits[*ap >> 4];
        ((struct arpcom *)ifp)->ac_enaddr[5] = rng >> 8;
                        crc >>= 1;
                        c >>= 1;
                        crc <<= 1;
                        c >>= 1;
                crc = (crc >> 4) ^ crctab[crc & 0xf];
                crc = (crc >> 4) ^ crctab[crc & 0xf];
                crc = (crc << 4) ^ crctab[(crc >> 28) ^ rev[data & 0xf]];
                crc = (crc << 4) ^ crctab[(crc >> 28) ^ rev[data >> 4]];
mimas$
```

Each line performs a bitwise operation, but it looks so cryptic that we'd rather leave it alone right? Well you'll probably encounter this a lot in C and especially in drivers (in the kernel).

- `& is bitwise AND`
- `| is bitwise OR`
- `^ is bitwise XOR`
- `! is bitwise NOT`
- `>> is bitwise rightshift`
- `<< is bitwise leftshift`
- `~ is unary (one's complement)`

Flags are often used. Like in the /sbin/ifconfig command:

```
mimas$ ifconfig em1
em1: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
```

The interface is UP, broadcasts, is running, does simplex and multicasting. These flags are defined in /usr/include/net/if.h and look like this:

```
#define IFF_UP          0x1             /* interface is up */
#define IFF_BROADCAST   0x2             /* broadcast address valid */
#define IFF_DEBUG       0x4             /* turn on debugging */
#define IFF_LOOPBACK    0x8             /* is a loopback net */
#define IFF_POINTOPOINT 0x10            /* interface is point-to-point link */
#define IFF_NOTRAILERS  0x20            /* avoid use of trailers */
#define IFF_RUNNING     0x40            /* resources allocated */
#define IFF_NOARP       0x80            /* no address resolution protocol */
#define IFF_PROMISC     0x100           /* receive all packets */
....
```

Notice the pattern which they are defined with, 0x1, 0x2, 0x4, 0x8, 0x10,

0x20, 0x40, 0x80, 0x100 and so on. This is hexadecimal because it starts with "0x" and is also exponential so that each value would be represented by each bit in an integer. Now let's put the bitwise operators to use...

```
u_int flags;
flags = 0;
flags |= IFF_UP; /* this sets the bit 0x1 to on */
```

As you can see we did flags = flags | 0x1, or flags is 0 OR 1 which is 1. Consider these truth tables to help you with bitwise operation

```
AND   |  0 |  1           OR |  0 |   1          XOR |  0  |  1
------+----+-----       -----+----+-----        -----+-----+-----
   0  |  0 |  0            0 |  0 |   1          0   |  0  |  1
------+----+-----       -----+----+-----        -----+-----+-----
   1  |  0 |  1            1 |  1 |   1          1   |  1  |  0
```

The way the truth tables are read are the top row and the left most column have a 1 and a 0 for on and off given the operation (AND, OR, XOR) the 2 numbers are then applied against each other and the result gets written in the middle where both cross in the column and row.

Back to our OR operation to turn on the flag, if I want to turn off that flag I would write the following in C

```
flags &= ~IFF_UP;  /* this sets bit 0x1 off */
```

as you can see it's flags = flags & ~0x1.. this basically takes the ones complement of 1 which is all bits turned on BUT 1 and then ANDS that to the existing flags because only bits that are on remain on when ANDED the bit 1 is turned off because it is 0 due to the unary.

Doing checks to see if flag is set would be done with a simple AND like:

```
if ((flags & IFF_UP) == IFF_UP)
 printf("UP ");
```

On to bitwise shift operations. If you have a value, let's call it crc and it is 0x1 then when we left shift it by 1 it becomes 0x2:

```
u_int crc = 0x1;
crc <<= 1;  /* value is now 0x2 because crc = crc << 1; */
crc >>= 2;  /* right shift by two the value is now 0 because the set bit falls off the end */
```

Left and right shifting is used in cryptographic operations often (as well as XOR) where the intention is to scramble the output in a manner that is reversable with a key (symmetric cryptography).

The XOR (^) is used in a lot of cryptographic operations, one such

cryptographic operations is a one time pad which is basically a stream of data XOR'ed with another secret and random stream of data (the pad). As long as the stream of data for the pad does not repeat itself (EVER!) the one time pad is secure.

You may ask what the power-of sign in C is then if ^ is taken which is often used as an exponential power sign, it is called a function pow() in math.h, ^ is XOR in C not exponents.

Take this snippet from aes_core.c in the OpenSSL crypto library, recognize any of the bitwise operations?

```
...
if (bits == 128) {
        while (1) {
                temp  = rk[3];
                rk[4] = rk[0] ^
                        (Te2[(temp >> 16) & 0xff] & 0xff000000) ^
                        (Te3[(temp >>  8) & 0xff] & 0x00ff0000) ^
                        (Te0[(temp      ) & 0xff] & 0x0000ff00) ^
                        (Te1[(temp >> 24)       ] & 0x000000ff) ^
                        rcon[i];
....
```

Yes I'm sure you will now. I see XOR, rightshift and AND operations. XOR is extensively used in crypto because it is reversable 0 ^ 1 == 1 and 1 ^ 1 == 0 thus the same key can turn crypt/decrypt an output. However there must be other permutations otherwise an XOR is easily broken.

13. Internet programming

We're now going to dive into network programming. This is a complex topic because a client needs a server and vice versa. We're going to just program a client first that connects the the TCP daytime port with a specified IP address. This is programmed in IPv4 programming so it's a little outdated, but it makes a good example. A client uses the socket system call to aquire a socket descriptor and the connect system call to connect to the remote address. Here is an example of how it would look.

```
dione$ ./tcptest 127.0.0.1
trying 127.0.0.1...
The time of day on 127.0.0.1 is Sat Apr 30 18:01:31 2011
dione$
```

And here is the source code:

```
     1  #include <sys/types.h>
     2  #include <sys/socket.h>
     3
     4  #include <netinet/in.h>
     5  #include <arpa/inet.h>
     6
     7  #include <netdb.h>
     8
     9  #include <stdio.h>
    10  #include <stdlib.h>
    11  #include <string.h>
    12
    13  int
    14  main(int argc, char *argv[])
    15  {
    16          struct sockaddr_in sin;
    17          int so, len;
    18          char buf[512];
    19
    20          if (argc != 2) {
    21                  fprintf(stderr, "usage: ./tcptest [ip address]\n");
    22                  exit(1);
    23          }
    24
    25          so = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    26          if (so < 0) {
    27                  perror("socket");
    28                  exit(1);
    29          }
    30
    31          memset(&sin, 0, sizeof(sin));           /* fill with 0's */
    32          sin.sin_family = AF_INET;
    33          sin.sin_port = htons(13);               /* daytime port */
    34          sin.sin_addr.s_addr = inet_addr(argv[1]);
    35
    36          printf("trying %s...\n", argv[1]);
    37
    38          if (connect(so, (struct sockaddr *)&sin, sizeof(sin)) < 0) {
```

```
39                  perror("connect");
40                  exit(1);
41          }
42
43          len = recv(so, buf, sizeof(buf) - 1, 0);
44
45          while (len > 0) {
46                  buf[len] = '\0';
47                  printf("The time of day on %s is %s", argv[1], buf);
48                  len = recv(so, buf, sizeof(buf) - 1, 0);
49          }
50
51          close(so);
52          exit(0);
53  }
```

* Line 1 through 7 are the includes for network code on a UNIX like operating system.
* Line 25 assigns the integer so the descriptor returned from the socket call, which requests the descriptor for address family INET (IPv4), socket of type STREAM (TCP) and protocol of IPPROTO_TCP (TCP).
* Line 26 checks for socket failure and bails out with exit code 1 if so is -1.
* Line 31 fills the struct sockaddr_in sin with zeros, this is necessary to make sure all members are zeroed, it's a common beginner error not to zero this struct and then errors would show up.
* Line 31-34 fills the struct with address family, port and IP address, which is converted to an integer with the function inet_addr() from a string.
* Line 38 tries to connect to the remote address where we pass sin as the second argument but cast it to struct sockaddr pointer.
* On line 43 we receive our first 511 or lower bytes from the remote IP and assign the length to the integer len.
* The subsequent recv() on line 48 should return 0 because the server will disconnect the session thus breaking the loop, finally we close the descriptor on line 51 and exit with successful status 0.

We'll now rewrite the same program to use IPv6. Not only that but we'll also introduce you to the getaddrinfo() function which is extremely helpful. We won't explore the entire functionality of getaddrinfo(), that's up to you to do by reading the manual page of it. We're just interested in connecting to IPv6's daytime port.

```
 1  #include <sys/types.h>
 2  #include <sys/socket.h>
 3  #include <netinet/in.h>
 4  #include <arpa/inet.h>
 5  #include <netdb.h>
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include <string.h>
 9
10  int
11  main(int argc, char *argv[])
12  {
13          int so, len, error;
14          char buf[512];
15          struct addrinfo hints, *res0;
16
17
18          if (argc != 2) {
19                  fprintf(stderr, "usage: ./gai_tcptest address\n");
20                  exit(1);
21          }
```

```
    22
    23          memset(&hints, 0, sizeof(hints));
    24          hints.ai_family = AF_INET6;
    25          hints.ai_socktype = SOCK_STREAM;                /* tcp */
    26
    27          error = getaddrinfo(argv[1], "daytime", &hints, &res0);
    28          if (error) {
    29                  fprintf(stderr, "getaddrinfo: %s\n",
gai_strerror(error));
    30                  exit(1);
    31          }
    32
    33          if (res0 != NULL) {
    34                  so = socket(res0->ai_family, res0->ai_socktype, res0-
>ai_protocol);
    35                  if (so < 0) {
    36                          perror("socket");
    37                          exit(1);
    38                  }
    39
    40                  if (connect(so, res0->ai_addr, res0->ai_addrlen) < 0) {
    41                          perror("connect");
    42                          exit(1);
    43                  }
    44          }
    45
    46          freeaddrinfo(res0);
    47
    48          len = recv(so, buf, sizeof(buf) - 1, 0);
    49
    50          while (len > 0) {
    51                  buf[len] = '\0';
    52                  printf("The time of day on %s is %s", argv[1], buf);
    53                  len = recv(so, buf, sizeof(buf) - 1, 0);
    54          }
    55
    56          close(so);
    57          exit(0);
    58  }
```

* Lines 23 through 30 show the getaddrinfo() calls. It has a hints structure to pass information to it (in order to keep the arguments short). res0 is then filled with data related to the connection.

Here is how this will look like:

```
dione$ ./gai_tcptest ::1
The time of day on ::1 is Sun May  1 17:56:14 2011
dione$ ./gai_tcptest localhost
The time of day on localhost is Sun May  1 17:56:19 2011
```

Notice that we can now use hostnames as getaddrinfo resolves. We could also have getaddrinfo try both IPv4 and IPv6 depending on which one would work. That info is in the manpage.

We'll now write a server. Because there is a lot behind writing a server this source code is a bit lengthy. We'll build a forking server which is easiest of

all, with least amount of code. Here it is...

```
 1  #include <sys/types.h>
 2  #include <sys/socket.h>
 3  #include <sys/wait.h>
 4  #include <sys/time.h>
 5  #include <netinet/in.h>
 6  #include <arpa/inet.h>
 7  #include <netdb.h>
 8  #include <unistd.h>
 9  #include <stdio.h>
10  #include <stdlib.h>
11  #include <string.h>
12  #include <signal.h>
13
14  #define MAXSOCK 32
15
16  void onchld(int sig);
17
18  int
19  main(void)
20  {
21          struct addrinfo hints, *res, *res0;
22          struct sockaddr_storage sto;
23          int so[MAXSOCK], error;
24          int nsock, maxsock, nso, i;
25          socklen_t stolen;
26          pid_t pid;
27          fd_set rset;
28          char *p, *message = "Thu Jan  1 01:00:00 CET 1970\r\n";
29
30          memset(&hints, 0, sizeof(hints));
31          hints.ai_family = PF_UNSPEC;
32          hints.ai_socktype = SOCK_STREAM;
33          hints.ai_flags = AI_PASSIVE;
34
35          error = getaddrinfo(NULL, "daytime", &hints, &res0);
36          if (error) {
37         fprintf(stderr, "getaddrinfo: %s", gai_strerror(error));
38                  exit(1);
39          }
40
41          nsock = 0;
42   for (res = res0; res && nsock < MAXSOCK; res = res->ai_next) {
43            so[nsock] = socket(res->ai_family, res->ai_socktype,
44                          res->ai_protocol);
45                  if (so[nsock] < 0) {
46                          perror("socket");
47                          exit(1);
48                  }
49        if (bind(so[nsock], res->ai_addr, res->ai_addrlen) < 0) {
50                          perror("bind");
51                          exit(1);
52                  }
53                  listen(so[nsock], 5);
54                  nsock++;
55          }
56
57          if (nsock == 0) {
```

```
 58        fprintf(stderr, "couldn't find interfaces to bind to\n");
 59                exit(1);
 60        }
 61
 62        freeaddrinfo(res0);
 63        daemon(0,0);
 64        signal(SIGCHLD, onchld);
 65
 66        for (;;) {
 67                FD_ZERO(&rset);
 68                for (i = 0; i < nsock; i++) {
 69                        FD_SET(so[i], &rset);
 70                        if (maxsock < so[i])
 71                                maxsock = so[i];
 72                }
 73
 74          error = select(maxsock + 1, &rset, NULL, NULL, NULL);
 75                if (error == -1)
 76                        continue;
 77
 78                for (i = 0; i < nsock; i++) {
 79                        if (FD_ISSET(so[i], &rset)) {
 80                  stolen = sizeof(struct sockaddr_storage);
 81              if ((nso = accept(so[i], (struct sockaddr *)&sto,
 82                                        &stolen)) < 0) {
 83                                        continue;
 84                                }
 85
 86                                switch (pid = fork()) {
 87                                case -1:
 88                                        continue;
 89                                case 0:
 90                                     for (p =message;*p;p++) {
 91                                            send(nso, p, 1, 0);
 92                                        }
 93                                        close(nso);
 94                                        exit(0);
 95
 96                                default:
 97                                        close(nso);
 98                                        break;
 99                                } /* switch .. */
100                        } /* if .. */
101                } /* for .. */
102        } /* for .. */
103        /* NOTREACHED */
104        exit(1);
105  }
106
107  void
108  onchld(int sig)
109  {
110        pid_t pid;
111        int status;
112
113        for(;;) {
114                pid = waitpid(0, &status, WNOHANG);
115
116                if (pid <= 0)
117                        return;
118        }
```

```
119  }
```

* Lines 1 to 12 are the appropriate include files
* Line 14 defines MAXSOCK to 32, a large number
* Line 16 defines the prototype of onchld() a signal handler
* Lines 30 through 33 fill the getaddrinfo hints with AI_PASSIVE flag to indicate that this is a server.
* Lines 42 through 55 perform (socket, bind, listen) the syscalls to set up a TCP server, repeat this for all address families as spit back from res0.
* Line 63 calls daemon() to send this process into the background
* Line 64 installs a signal handler for the SIGCHLD (child) signal.
* Lines 67 through 72 add the active sockets to a select mask so that we can accept on those descriptors that are ready, otherwise accept() will block and cause a DoS.
* Lines 74 is the select() to only let through the readable sets
* Line 81 accepts the new connection with the accept() system call, this is the last syscall needed for a server.
* Line 86 through 99 fork a new process and in the child we write byte for byte into the network, we do this for a reason watch explanation later. Once the message has been written we close the descriptor and exit the child.
* Lines 108 through 119 are a signal handler to wait on zombie childs which are signaled with the installed SIGCHLD signal call. This is important or else we have a lot of processes in the Z state.

So now we have this server running (we disabled the inetd daytime in order to bind to port 13). We can telnet to it:

```
# telnet ::1 daytime
Trying ::1...
Connected to ::1.
Escape character is '^]'.
Thu Jan  1 01:00:00 CET 1970
Connection closed by foreign host.
```

It looks just like the other daytime services, but there is a difference, watch. We'll now use our made programs and use these on this daytime service:

```
# ./tcptest 127.0.0.1
trying 127.0.0.1...
The time of day on 127.0.0.1 is TThe time of day on 127.0.0.1 is hu Jan  1 01:00:00 CET 1970
# ./gai_tcptest ::1
The time of day on ::1 is TThe time of day on ::1 is hu Jan  1 01:00:00 CET 1970
#
```

The output is a lot different from the original tests and there is an explanation. This server sends the message byte for byte with the send() system call. The OS detects that there is a byte for byte send and executes what's called a nagle algorithm so that all these bytes can be grouped. However the first makes it out and messes up our input. The clients themselves only expected everything to arrive in one packet and thus they loop around the message's bytes.

Here then is how an improvement may look like on part of the client, the server's fix is easy if we want to go that route, but we don't:

```
1  #include <sys/types.h>
2  #include <sys/socket.h>
```

```
  3  #include <netinet/in.h>
  4  #include <arpa/inet.h>
  5  #include <netdb.h>
  6  #include <stdio.h>
  7  #include <stdlib.h>
  8  #include <string.h>
  9
 10  int
 11  main(int argc, char *argv[])
 12  {
 13          int so, len, error;
 14          char buf[512], *p;
 15          struct addrinfo hints, *res0;
 16
 17
 18          if (argc != 2) {
 19              fprintf(stderr, "usage: ./gai_tcptest2 address\n");
 20                  exit(1);
 21          }
 22
 23          memset(&hints, 0, sizeof(hints));
 24          hints.ai_family = AF_INET6;
 25       hints.ai_socktype = SOCK_STREAM;                /* tcp */
 26
 27          error = getaddrinfo(argv[1], "daytime", &hints, &res0);
 28          if (error) {
 29   fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(error));
 30                  exit(1);
 31          }
 32
 33          if (res0 != NULL) {
 34 so = socket(res0->ai_family, res0->ai_socktype, res0->ai_protocol);
 35                  if (so < 0) {
 36                          perror("socket");
 37                          exit(1);
 38                  }
 39
 40                  if (connect(so, res0->ai_addr,res0->ai_addrlen) < 0) {
 41                          perror("connect");
 42                          exit(1);
 43                  }
 44          }
 45
 46          freeaddrinfo(res0);
 47
 48          p = &buf[0];
 49          len = recv(so, p++, 1, 0);
 50
 51          while (len > 0 && (p - &buf[0]) < 511) {
 52                  len = recv(so, p++, 1, 0);
 53          }
 54          buf[(p - &buf[0]) - 1] = '\0';
 55          printf("The time of day on %s is %s", argv[1], buf);
 56
 57          close(so);
 58          exit(0);
 59  }
```

Output now looks like this:

```
# ./gai_tcptest2 ::1
The time of day on ::1 is Thu Jan  1 01:00:00 CET 1970
```

14. External Libraries

So far we've been dealing with C code that is in libc (library for C). (With exception for the TCP code which needs libraries nsl and socket in Solaris). This is the builtin library that the C compiler defaults to. You can see it as a file in the /usr/lib directory:

```
dione$ ls -l libc.*
-r--r--r--  1 root  bin  6889318 Aug 16  2010 libc.a
-r--r--r--  1 root  bin  3072519 Aug 16  2010 libc.so.56.0
```

.a means it's an archive (another name for library) and .so.56.0 means that part is for the shared libary used for dynamic linking.

There is other libraries too that are useful for making complex code easier for the programmer. Pretend you compiled a program that requires a library the normal way you'd see this:

```
dione$ cc -o mygunzip mygunzip.c
/tmp//ccGfnM1J.o(.text+0x65): In function `main':
: undefined reference to `gzopen'
/tmp//ccGfnM1J.o(.text+0xbc): In function `main':
: undefined reference to `gzread'
/tmp//ccGfnM1J.o(.text+0xd7): In function `main':
: undefined reference to `gzclose'
collect2: ld returned 1 exit status
dione$
```

To link this libary that's missing into our code we have to know what library these functions belong to. A manpage lookup on gzopen shows us

```
COMPRESS(3)              OpenBSD Programmer's Manual             COMPRESS(3)

NAME
    compress - zlib general purpose compression library
```

So chances are that the library that it belongs to is called "zlib" (this is just intuition). We make sure that it exists by ls'ing in the libraries:

```
dione$ ls -l /usr/lib/libz.*
-r--r--r--  1 root  bin  299084 Aug 16  2010 /usr/lib/libz.a
-r--r--r--  1 root  bin  201372 Aug 16  2010 /usr/lib/libz.so.4.1
```

And to really make sure we can "nm" the library for the function, like so:

```
dione$ nm libz.a|grep gzopen
00001670 T gzopen
```

So to link this library into our source code we use the flag -l to the compiler:

```
dione$ cc -o mygunzip mygunzip.c -lz
dione$
```

Success! Let's see if the program works.

```
dione$ cp /bin/sh /tmp
dione$ gzip /tmp/sh
dione$ ./mygunzip /tmp/sh.gz | md5
149a75c794270a5e0a67d94d32709799
dione$ md5 /bin/sh
MD5 (/bin/sh) = 149a75c794270a5e0a67d94d32709799
```

works perfectly. For reference you can have the source code for this, it's here:

```
     1  #include <sys/types.h>
     2  #include <unistd.h>
     3  #include <stdio.h>
     4  #include <stdlib.h>
     5  #include <string.h>
     6
     7  #include <zlib.h>
     8
     9  int
    10  main(int argc, char *argv[])
    11  {
    12          gzFile *gzfd;
    13          int len;
    14          char buf[512];
    15
    16          if (argc != 2)  {
    17                  fprintf(stderr, "usage: ./mygunzip <file.gz>\n");
    18                  exit(1);
    19          }
    20
    21          gzfd = gzopen(argv[1], "r");
    22          if (gzfd == NULL) {
    23                  perror("gzopen");
    24                  exit(1);
    25          }
    26          while ((len = gzread(gzfd, buf, sizeof(buf))) > 0) {
    27                  write(STDOUT_FILENO, buf, len);
    28          }
    29
    30          gzclose(gzfd);
    31          exit(0);
    32  }
```

15. Still doesn't cut it?

While I was writing this document, I did not go into things such as http://en.wikipedia.org/wiki/Binary_search, http://en.wikipedia.org/wiki/Hash_function or http://en.wikipedia.org/wiki/B-Tree. Nor sorting which is needed to access these methods. These are invaluable methods in programming in that they reduce execution time immensely. The K&R book mentioned at the top goes into this and it shouldn't be overlooked. Also one book that I found invaluable was a follow-up to K&R called "The Practice of Programming" by Brian W. Kernighan and Rob Pike. I use this book for the formulas in the first few chapters.

16. Other Online Tutorials

If you're looking for more works for free online tutorials there is finished and works in progress, I'll list some here:

- Introduction to C http://programming.coreth.com/c/
- Beej's Guide to Network Programming http://beej.us/guide/bgnet/output/html/singlepage/bgnet.html
- The C Book http://publications.gbdirect.co.uk/c_book/

17. Bibliography

- The (ANSI) C Programming Language - Brian Kernighan and Dennis Ritchie
- The UNIX programming environment - Brian Kernighan and Rob Pike