Unix/C Notes

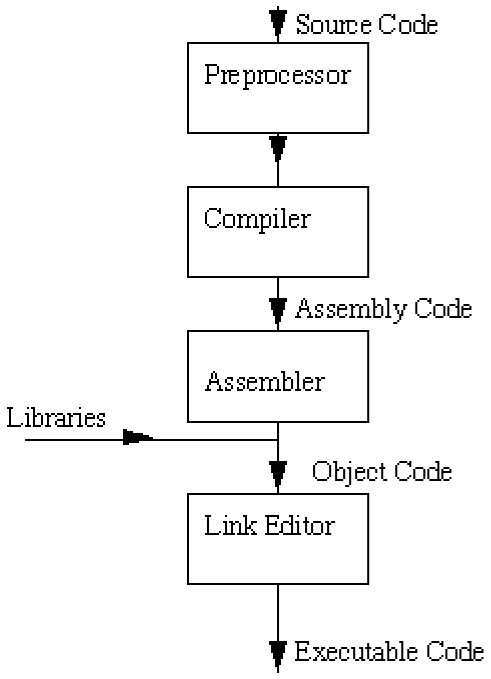
Compiled by:
Watsh Rajneesh
Software Engineer @ Quark (R&D Labs)
wrajneesh@bigfoot.com

References

1. http://www.yendor.com/programming/unix/apue/app-c.html -- Solutions to Richard Stevens' masterpiece on Advanced Programming in Unix Environment.
2. http://www.cs.cf.ac.uk/Dave/C/CE.html -- Good C programming (for UNIX OS) reference.
3. http://www.erlenstar.demon.co.uk/unix/faq_toc.html -- Unix FAQ.

Contents

1. C Programming in Unix environment



The figure above shows the **C compilation model.**

Preprocessor -- The Preprocessor accepts source code as input and is responsible for removing comments and interpreting special preprocessor directives denoted by #.
C Compiler -- translates source to assembly.
Assembler -- creates object code.
Link Editor -- If a source file references library functions or functions defined in other source files the link editor combines these functions (with main()) to create an executable file. External Variable references resolved here also.
Useful **compiler options**:
gcc [option | filename]...
g++ [option | filename]...

-c : Disable linking. Later all the object files can be linked as,

```
gcc file1.o file2.o ...... -o executable
```

-llibrary : Link with object libraries.

```
gcc calc.c -o calc -lm
```

-Ldirectory : Add directory to the list of directories containing object-library routines.The linker always looks for standard and other system libraries in /lib and /usr/lib. If you want to link in libraries that you have created or installed yourself (unless you have certain privileges and get the libraries installed in **/usr/lib**) you will have to specify where you files are stored, for example:

```
    gcc prog.c -L/home/myname/mylibs mylib.a
```

-Ipathname : Add pathname to the list of directories in which to search for #include files with relative filenames (not beginning with slash /).
BY default, The preprocessor first searches for #include files in the directory containing source file, then in directories named with -I options (if any), and finally, in **/usr/include**.
So to include header files stored in /home/myname/myheaders you would do:

```
    gcc prog.c -I/home/myname/myheaders
```

Note: System library header files are stored in a special place (/usr/include) and are not affected by the -I option. System header files and user header files are included in a slightly different manner

-g : invoke debugging option. This instructs the compiler to produce additional symbol table information that is used by a variety of debugging utilities.

-D : define symbols either as identifiers (-Didentifer) or as values (-Dsymbol=value) in a similar fashion as the #define preprocessor command.

Explore the libraries to see what each contains by running the command **ar t** libfile.
man 3 ctime   -- section 3 of the unix manual contains documentation of the standard c library functions.
cat <filename> | more -- view a file.

---

2. Advanced C topics

Some useful tables for C Programming are given below:

FORMAT STRING	DESCRIPTION	scanf()
	Numeric Values	
%d	Decimal integer	y
%f	Floating point decimal(Real number)	y
%lf	Double (Long Real number)	y
%e	Exponential representation	n
%u	Unsigned (positive) integer	n
%x	Hexadecimal integer	y
%o	Octal integer	y
%g	Automatically Selects Shorter of %f and %e	n
	Character Types	
%c	Single character	y
%s	Character String, Defined as char *	y

C code	Meaning
`'\014'`	Bit pattern for Form Feed
`'\n'`	Newline
`'\t'`	Tab
`'\\ '`	Backslash
`'\''`	Single Quote
`'\"'`	Double Quote
`'\b'`	Backspace
`'\r'`	Carriage Return
`'\f'`	Form Feed
`'\0'`	NULL (String Terminator)

Table: Formatted I/O
Specifiers. Table:
Special characters.

	Description	Represented By
1	Parenthesis	`() []`
1	Structure Access	`. ->`
2	Unary	`! ~ ++ -- - * &`
3	Mutiply, Divide, Modulus	`* / %`
4	Add, Subtract	`+ -`
5	Shift Right, Left	`>> <<`
6	Greater, Less Than, etc	`> < =`
7	Equal, Not Equal	`== !=`
8	Bitwise AND	`&`
9	Bitwise Exclusive OR	`^`
10	Bitwise OR	`|`
11	Logical AND	`&&`
12	Logical OR	`||`
13	Conditional Expression	`?:`
14	Assignment	`= += -=` etc
15	Comma	`,`

Table: Precedence of C Operators.

Note: Unary, Assignment and Conditional operators associate Right to Left. Others associate from Left to Right.

2.1 Dynamic memory allocation

Malloc	`void *malloc(size_t number_of_bytes);` That is to say it returns a pointer of type void * that is the start in memory of the reserved portion of size number_of_bytes. If memory cannot be allocated a NULL pointer is returned.

	Since a void * is returned the C standard states that this pointer can be converted to any type. The size_t argument type is defined in stdlib.h and is an unsigned type. `int* ip = (int *) malloc(100*sizeof(int));` When you have finished using a portion of memory you should always free() it. This allows the memory freed to be aavailable again, possibly for further malloc() calls The function **free()** takes a pointer as an argument and frees the memory to which the pointer refers.

Calloc	**`void *calloc(size_t num_elements, size_t element_size};`**
	Malloc does not initialise memory (to zero) in any way. If you wish to initialise memory then use calloc. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num_elements, and element_size, element_size, as two individual arguments.
	int* ip = (int *) calloc(100, sizeof(int));

Realloc	**`void *realloc( void *ptr, size_t new_size);`** Realloc is a function which attempts to change the size of a previous allocated block of memory. The new size can be larger or smaller. If the block is made larger then the old contents remain unchanged and memory is added to the end of the block. If the size is made smaller then the remaining contents are unchanged. If the original block size cannot be resized then realloc will attempt to assign a new block of memory and will copy the old block contents. Note a new pointer (of different value) will consequently be returned. You must use this new value. If new memory cannot be reallocated then realloc returns NULL. Thus to change the size of memory allocated to the *ip pointer above to an array block of 50 integers instead of 100, simply do: ip = (int *) calloc(ip, 50);

See also: Data Structures in C/C++ notes.

2.2 Low level bitwise operators and bit fields

Bitwise operators:

Many programs (e.g. systems type applications) must actually operate at a low level where individual bytes must be operated on. The combination of pointers and bit-level operators makes C useful for many low level applications and can almost replace assembly code. (Only about 10 % of UNIX is assembly code the rest is C!!.)

Table: Bitwise operators

&	AND
|	OR

^	XOR
~	1's complement
<<	Left Shift
>>	Right Shift

if x = 00000010 (binary) or 2 (decimal) then:
x>>2 => x = 00000000
x<<2 => x = 00001000
~x = 11111111
A shift left is equivalent to a multiplication by 2. Similarly a shift right is equal to division by 2 Shifting is much faster than actual multiplication (*) or division (/) by 2. So **if you want fast multiplications or division by 2 use shifts**.
To illustrate many points of bitwise operators let us write a function, Bitcount, that counts bits set to 1 in an 8 bit number (unsigned char) passed as an argument to the function.

```
int BitCount(unsigned char x) {
    int count;
    for(count = 0;x != 0;x >>= 1) {
        if(x & 01) count++;
    }
    return count;
}
```

This function illustrates many C program points:

for loop not used for simple counting operation
x>>1 => x = x >> 1
for loop will repeatedly shift right x until x becomes 0
use expression evaluation of x & 01 to control if
x & 01 masks of 1st bit of x if this is 1 then count++

Bit Fields:
Bit Fields allow the **packing of data in a structure**. This is especially useful when memory or data storage is at a premium. Typical examples:
Packing several objects into a machine word. e.g. 1 bit flags can be compacted -- Symbol tables in compilers.
Reading external file formats -- non-standard file formats could be read in. E.g. 9 bit integers.
C lets us do this in a structure definition by putting :bit length after the variable. i.e.

```
struct packed_struct {
   char* name;
   unsigned f1:1;
   unsigned f2:1;
   unsigned f3:1;
   unsigned f4:1;
   unsigned type:4;
   unsigned funny_int:9;
} pack;
```

Here the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4 bit type and a 9 bit funny_int. C automatically packs the above bit fields as compactly as possible, provided that **the maximum length of the field is less than or equal to the integer word length of the computer**. If this is not the case then some compilers may allow memory overlap for the fields whilst other would store the next field in the next word But for portability, as a thumb rule, the fields in the above struct must be kept of maximum 32 bits for a machine with 32 bit word length. The important points about bit fields are:

Access members as usual via: pack.type = 7;
Only n lower bits will be assigned to an n bit number. So type cannot take values larger than 15 (4 bits long).
Bit fields are always converted to integer type for computation.
You are allowed to mix ``normal'' types with bit fields.
The unsigned definition is important - ensures that no bits are used as a +/- flag
Bit fields can be used to save space in structures having several binary flags or other small fields. They can also be used in an attempt to conform to externally imposed storage layout. Their success at the later task is mitigated by the fact that bitfields are assigned left to right on some machines (big-endian) and right to left on others (little-endian). Note that the colon notation for specifying the size of a field in bits is valid only in structures (and in unions).

Bit fields can be **signed** or **unsigned**. Plain bit fields are treated as **signed**.Bit fields are allocated within an integer from least-significant to most-significant bit. In the following code

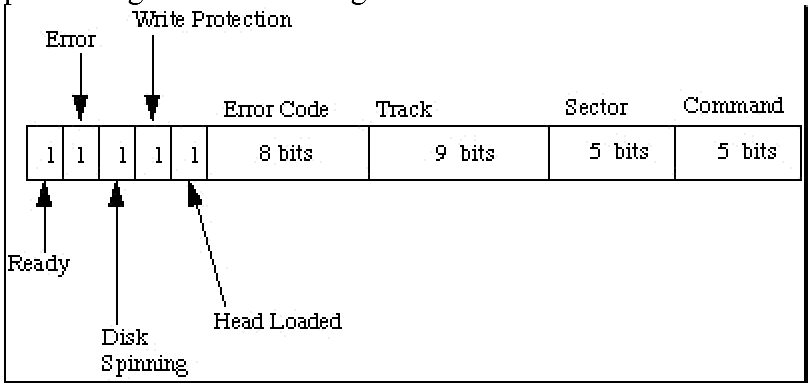
```
struct mybitfields
{
   unsigned a : 4;
   unsigned b : 5;
   unsigned c : 7;
} test;

void main( void )
{
   test.a = 2;
   test.b = 31;
   test.c = 0;
}
```

the bits would be arranged as follows:

```
00000001 11110010
cccccccb bbbbaaaa
```

Since the 80x86 processors store the low byte of integer values before the high byte, the integer 0x01F2 above would be stored in physical memory as 0xF2 followed by 0x01.

Practical Example for Bit Fields:
Frequently device controllers (e.g. disk drives) and the operating system need to communicate at a low level. Device controllers contain several registers which may be packed together in one integer.



We could define this register easily with bit fields:

```
struct DISK_REGISTER  {
     unsigned ready:1;
     unsigned error_occured:1;
     unsigned disk_spinning:1;
     unsigned write_protect:1;
     unsigned head_loaded:1;
     unsigned error_code:8;
     unsigned track:9;
     unsigned sector:5;
     unsigned command:5;
};
```

To access values stored at a particular memory address, DISK_REGISTER_MEMORY we can assign a pointer of the above structure to access the memory via: `struct DISK_REGISTER *disk_reg = (struct DISK_REGISTER *) DISK_REGISTER_MEMORY;`

The disk driver code to access this is now relatively straightforward:

```
/* Define sector and track to start read */

disk_reg->sector = new_sector;
disk_reg->track = new_track;
disk_reg->command = READ;

/* wait until operation done, ready will be true */

while ( ! disk_reg->ready ) ;

/* check for errors */

if (disk_reg->error_occured)  {
    /* interrogate disk_reg->error_code for error type */
    switch (disk_reg->error_code)
    ......
  }
```

Padding and Alignment of structure members:
Structure members are stored sequentially in the order in which they are declared: the first member has the lowest memory address and the last member the highest.

Every data object has an alignment-requirement. The alignment-requirement for all data except structures, unions, and arrays is either the size of the object or the current packing size (specified with either /Zp or the pack pragma in VC++ 6.0, whichever is less). For structures, unions, and arrays, the alignment-requirement is the largest alignment-requirement of its members. Every object is allocated an offset so that

offset % alignment-requirement == 0

Adjacent bit fields are packed into the same 1-, 2-, or 4-byte allocation unit if the integral types are the same size and if the next bit field fits into the current allocation unit without crossing the boundary imposed by the common alignment requirements of the bit fields.
Some Exercises for this section: Solutions below have been tested to be working on MS VC++ 6.0.
1. Write a function that prints out an 8-bit (unsigned char) number in binary format.

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  unsigned char ch;
  int i;
  scanf("%d",&ch);
  for(i = 0; i < 8; i++) {

    if(ch & 01) {
        printf("1");
    }
    else {
      printf("0");
    }
    ch = ch >> 1;
   }
}
```

2. Write a function setbits(x,p,n,y) that returns x with the n bits that begin at position p set to the rightmost n bits of an unsigned char variable y (leaving other bits unchanged). E.g. if x = 10101010 (170 decimal) and y = 10100111 (167 decimal) and n = 3 and p = 6 say then you need to strip off 3 bits of y (111) and put them in x at position 10xxx010 to get answer 10111010. Your answer should print out the result in binary form. Your output should be like this:

```
x = 01010101 (binary)
y = 11100101 (binary)
setbits n = 3, p = 6 gives x = 01010111 (binary)
```

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>

void printBinary(unsigned char ch) {
    int i;
 for(i = 0; i < (sizeof(unsigned char)*8); i++) {
  if(ch & 01) {
   printf("1");
  }
  else {
   printf("0");
  }
  ch = ch >> 1;
 }
    printf(" (binary)\n");
}

void setbits(unsigned char* x,int p,int n,unsigned char y) {
     // strip off n bits of y
     int i;
     unsigned char yBitsMask;
     unsigned char cleanNBitsFromXMask;
     int* bitarr = (int*)malloc(sizeof(int)* n);

     // strip off n bits of y
     for(i = 0; i < n; i++,y >>= 1) {
           if(y & 01) {
                bitarr[i] = 1;
           }
           else {
                bitarr[i] = 0;
           }
     }
     cleanNBitsFromXMask = (pow((double)2,(double)8) - 1);
     printBinary(cleanNBitsFromXMask);
     for(i = 0; i < n; i++) {
           cleanNBitsFromXMask -= pow((double)2,(double)(p+i-1));
     }
     printBinary(cleanNBitsFromXMask);
     // clean n bits of x from position p
     *x = (*x) & cleanNBitsFromXMask;

     yBitsMask = 0;
     for(i = 0; i < n; i++) {
           yBitsMask += (pow((double)2,(double)(p+i-1)) * bitarr[i]);
     }
     printBinary(yBitsMask);
     // insert the n bits of y in x from position p
     *x = (*x) | yBitsMask;
}

int main(int argc, char *argv[])
{
    unsigned char x,y;
    int p,n;
    scanf("%d %d %d",&x,&p,&n);
    scanf(" ");
    scanf("%d",&y);
    printf("x = ");
    printBinary(x);
    printf("y = ");
    printBinary(y);
    setbits(&x,p,n,y);
    printf("setbits n = %d, p = %d gives x = ",n,p);
    printBinary(x);
    printf("\n");
}
```

3. Write a function that inverts the bits of an unsigned char x and stores answer in y. Your answer should print out the result in binary form. Your output should be like this:

   x = 10101010 (binary)
   x inverted = 01010101 (binary)

```
unsigned char invertbits(unsigned char x) {
 int i;
 unsigned char temp = 0;

 for(i = 0; i < 8; i++,x>>=1) {
  if(!(x & 01)) {
   temp += pow((double)2,(double)i);
  }
 }
 return temp;
}
```

Add this function to the above program to test it. Call printBinary() function to print the o/p.

4. Write a function that rotates (**NOT** shifts) to the right by n bit positions the bits of an unsigned char x.ie no bits are lost in this process. Your answer should print out the result in binary form.Your output should be like this:

   x = 10100111 (binary)
   x rotated by 3 = 11110100 (binary)

```
void rotateBitsToRight(unsigned char* x, int times) {
 int i, toset = 0;
 for(i = 0; i < times; i++) {
  // remember whether to shift or not
  if((*x) & 01) {
   toset = 1;
  }
  else {
   toset = 0;

  }
  // cause the right shift
  *x = (*x) >> 1;
```

```
    // copy the shifted out bit at the last position
    if(toset) {
     *x = (*x) | 0x80; // set the last bit
    }
    else {
     *x = (*x) & 0x7F; // unset the last bit
    }
  }
}
```

Again you can test this with the above program.

2.3 Preprocessors

Preprocessor Directives	

#define	Use this to **define constants or any macro substitution**. Use as follows:
	    #define <macro> <replacement name>
	
	For Example:
	    #define FALSE 0
	    #define TRUE !FALSE
	
	We can also define small ``functions" using #define. For example max. of two variables:
	   #define max(A,B) ((A) > (B) ? (A):(B))
	So if in our C code we typed something like:
	   x = max(q+r,s+t);
	after preprocessing, if we were able to look at the code it would appear like this:
	   x = ((q+r) > (r+s) ? (q+r) : (s+t));

#undef	This commands **undefines a macro**. A macro must be undefined before being redefined to a different value.
#include	This directive includes a file into code. It has two possible forms: #include <file> or #include "file" <file> tells the compiler to look where system include files are held. Usually UNIX systems store files in usrinclude directory. "file" looks for a file in the current directory (where program was run from).Included files usually contain C prototypes and declarations from header files and not (algorithmic) C code. **Documentation of MSDN:** For file specifications enclosed in angle brackets, the preprocessor does not search directories of the parent files. A ôparentö file is the file that has the#include directive in it. Instead, it begins by searching for the file in the directories specified on the compiler command line following /I. If the /I option is not present or fails, the preprocessor uses the INCLUDE

	environment variable to find any include files within angle brackets. The INCLUDE environment variable can contain multiple paths separated by semicolons (;). If more than one directory appears as part of the /I option or within the INCLUDE environment variable, the preprocessor searches them in the order in which they appear. If you specify a complete, unambiguous path specification for the include file between two sets of double quotation marks (" "), the preprocessor searches only that path specification and ignores the standard directories. For include files specified as #include "path-spec", directory searching begins with the directories of the parent file, then proceeds through the directories of any grandparent files. Thus, searching begins relative to the directory containing the source file currently being processed. If there is no grandparent file and the file has not been found, the search continues as if the filename were enclosed in angle brackets.

#if	**#if -- E**valuates a constant integer expression. You always need a **#endif** to delimit end of statement. We can have else etc. as well by using **#else** and **#elif** -- else if. Another common use of #if is with:
	#ifdef
	-- if defined and
	#ifndef
	-- if not defined
	These are **useful for checking if macros are set** -- perhaps from different program modules and header files. For example, to set integer size for a portable C program between TurboC (on MSDOS) and Unix (or other) Operating systems. Recall that TurboC uses 16 bits/integer and UNIX 32 bits/integer. Assume that if TurboC is running a macro **TURBOC** will be defined. So we just need to check for this:
	`#ifdef TURBOC`
	`   #define INT_SIZE 16`
	`#else`
	`   #define INT_SIZE  32`
	`#endif`
	As another example if running program on MSDOS machine we want to include file msdos.h otherwise a default.h file. A macro **SYSTEM** is set (by OS) to type of system so check for this:
	`#if SYSTEM == MSDOS`
	`   #include <msdos.h>`
	`#else`
	` #include ``default.h''`
	`#endif`

#error *text of error message*	generates an appropriate compiler error message. e.g
	
	`#ifdef OS_MSDOS`
	`    #include <msdos.h>`
	`#elifdef OS_UNIX`
	` #include ``default.h''`

	`#else`
	`    #error Wrong OS!!`
	`#endif`

#line *number ["string"]*	informs the preprocessor that the number is the next number of line of input. "string" is optional and names the next line of input. This is most often used with programs that translate other languages to C. For example, error messages produced by the C compiler can reference the file name and line numbers of the original source files instead of the intermediate C (translated) source files.

Compiler control for preprocessor
You can use the cc compiler to control what values are set or defined from the command line. This gives some flexibility in setting customised values and has some other useful functions.
1. The **-D** compiler option is used. For example:

```
cc -DLINELENGTH=80 prog.c -o prog
```

has the same effect as:

```
#define LINELENGTH 80
```

Note that **any #define or #undef within the program (prog.c above) override command line settings**.

You can also set a symbol without a value, for example:

cc -DDEBUG prog.c -o prog

Here the value is assumed to be **1**.

The setting of such flags is useful, especially for debugging. You can put commands like:

```
#ifdef DEBUG
    print("Debugging: Program Version 1\");
#else
    print("Program Version 1 (Production)\");
#endif
```

Also since preprocessor command can be written anywhere in a C program you can filter out variables etc for printing etc. when debugging:

```
x = y *3;

#ifdef DEBUG
    print("Debugging: Variables (x,y) = \",x,y);
#endif
```

2. The **-E** command line is worth mentioning just for academic reasons. It is not that

practical a command. The -E command will force the compiler to stop after the preprocessing stage and output the current state of your program. Apart from being debugging aid for preprocessor commands and also as a useful initial learning tool it is not that commonly used.

Exercises:

1. Define a preprocessor macro swap(t,x, y) that will swap two arguments x and y of type t.

// To Do!

2. Define a preprocessor macro to select:

the least significant bit from an unsigned char

```
#define SELECT_LSB(x)  (((x) & 01) ? 1 : 0)
```

the nth (assuming least significant is 0) bit from an unsigned char.

```
#define SELECT_NTH_BIT(x,n)  ((x)>>(n),((x) & 01) ? 1 : 0)
```

2.4 C, Unix and Standard libraries

2.4.1 stdlib.h

To use all functions in this library you must:

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

There are three basic categories of functions:

- Arithmetic
- Random Numbers
- String Conversion

Arithmetic Functions

There are 4 basic integer functions:

```
int abs(int number);
long int labs(long int number);

div_t div(int numerator,int denominator);
ldiv_t ldiv(long int numerator, long int denominator);
```

Essentially there are two functions with integer and long integer compatibility.

abs functions return the absolute value of its number arguments. For example, abs(2) returns 2 as does abs(-2).

div takes two arguments, numerator and denominator and produces a quotient and a remainder of the integer division. The div_t structure is defined (in stdlib.h) as follows:

```
typedef struct {
      int  quot; /* quotient */
      int  rem;  /* remainder */
} div_t;
```

(ldiv_t is similarly defined).

Thus:

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

int num = 8, den = 3;
div_t ans;
ans = div(num,den);
printf("Answer:\n\t Quotient = %d\n\t Remainder = %d\n", ans.quot,ans.rem);
```

2.4.2 math.h
2.4.3 stdio.h
2.4.4 string.h
2.4.5 File access and Directory system calls
2.4.6 Time functions

3. Process control

4. General File Handling and IPC

4.1 Interrupts and Signals <signal.h>

The program below is described in the comments in Lines 3-15. It acts like the Unix grep command (which finds all lines in a file which contain the user-specified string), except that it is more interactive: The user specifies the files one at a time, and (here is why the signals are needed) he/she can cancel a file search in progress without canceling the grep command as a whole.

footnote: You should make sure that you understand why an ordinary scanf call will not work here.For example, when I ran the program (Line 2), I first asked it to search for the string `type' (Line 4). It asked me what file to search in, and I gave it a file name (Line 6); the program then listed for me (Lines 7-20) all the lines from the file HowToUseEMail.tex which contain the string `type'. Then the program asked me if I wanted to search for that (same) string in another file (Line 21), so I asked it to look in the system dictionary (Line 22). The program had already printed out the first few response lines (Lines 23-25) when I changed my mind and decided that I didn't want to check that file after all; so, I typed control-c (Line 26), and program responded by confirming that I had abandoned its search in that file (Line 26). I then gave it another file to search (Line 28).

```
 36  /* program to illustrate use of signals
 37
 38     this is like the Unix `grep' command; it will report all lines
 39     in a file that contain the given string; however, this one is
```

```
   different:  the user specifies the string first, and then the
   program prompts the user for the names of the files to check
   the string for, one file at a time

   reading a large file will take some time, and while waiting
   the user may change his/her mind, and withdraw the command
   to check this particular file (the given string is still
   valid, though, and will be used on whatever further files
   the user specifies); this is where the signals are used */



#define MaxLineLength 80


#include <stdio.h>
#include <signal.h>
#include <setjmp.h>


jmp_buf(GetFileName);


char Line[MaxLineLength],  /* one line from an input file */
     String[MaxLineLength],  /* current partial-word */
     FileName[50];  /* input file name */


int StringLength;  /* current length of the partial-word */


FILE *FPtr;


/* ReadLine() will, as its name implies, read one line of the file; it
   will return a value which will be the line length if it successfully
   reads a line, or -1 if the end of the file was encountered */

int ReadLine()

{  char C;  int LineLength;

   LineLength = 0;
   while (1)  {
      if (fscanf(FPtr,"%c",&C) == -1) return -1;
      if (C != '\n') Line[LineLength++] = C;
      else  {
         Line[LineLength] = 0;
         break;
      }
   }
   return LineLength;
}


```

```
 95   FindAllMatches()
 96
 97   {  int LL,J,MaxJ;
 98
 99      FPtr = fopen(FileName,"r");
100      while (1)  {
101         LL = ReadLine();
102         if (LL == -1) break;
103         MaxJ = LL - StringLength + 1;
104         for (J = 0; J < MaxJ; J++)
105            if (!strncmp(String,Line+J,StringLength))  {
106               printf("%s\n",Line);
107               break;
108            }
109      }
110   }
111
112
113   CtrlC()
114
115   {  printf("OK, forget that file\n");
116      longjmp(GetFileName);
117   }
118
119
120   main()
121
122   {  char NewLineChar;
123
124      signal(SIGINT,CtrlC);
125      printf("string?\n");  scanf("%s",String);
scanf("%c",&NewLineChar);
126      StringLength = strlen(String);
127      while (1)  {
128         setjmp(GetFileName);
129         printf("file name?\n");  scanf("%s",FileName);
130         if (!strcmp(FileName,"q")) exit();
131         FindAllMatches();
132      }
133   }
```

Analysis:

The non-signal part of the program is straightforward. The function main() has a while loop to go through each file (Lines 127-132), and for each file, there is a while loop to read in each line from the file and check for the given string (Lines 100-109).

footnote: Note the expression Line+J in Line 105. Recall that an array name without a subscript, in this case `Line', is taken to be a pointer to the beginning of that array. Thus Line+J is taken to be pointer arithmetic, with the result being a pointer to Line[J]; the string comparison of strncmp will begin there.
On Line 124 we have the call to signal(). **SIGINT** is the signal number for control-c signals; it is defined in the #include file mentioned above.

footnote: Or type man signal. There are lots of other signal types, e.g. **SIGHUP**, which is generated if the user has a phone-in connection to the machine and suddenly hangs up the phone while the program is running.
In this call to signal() we are saying that whenever the user types control-c, we want the program to call the function CtrlC() (Lines 113-117).
footnote: Such a function is called a **signal-handler**. We say, for example, that on Line 124 we are telling the system that we want our function CtrlC() to be the signal-handler for SIGINT-type signals.
When the user types control-c, we want the program to abandon the search in the present file, and start on the next file. In other words, when we finish executing CtrlC(), we do not want execution to resume at the line in the program where was at the instant the user typed control-c (typically somewhere in the range of Lines 100-109)--instead, what we want is for the program to jump to Line 129. This is accomplished by the longjmp() function, which in our case (Line 128) says to jump to the line named `GetFileName'. How do we assign a name to a line? Well, this is accomplished by the setjmp() function, which in our case (Line 128) says to name the next line (Line 129) GetFileName.

footnote: What is actually happening is that **GetFileName** will contain the memory address of the first machine instruction in the compiled form of Line 129. How can the function **setjmp**() ``know'' this address? The answer is that this address will be the return address on that stack at that time. By the way, a goto statement won't work here, because one can't jump to a goto which is in a different function.
(This ``name'' will actually be an integer array which records the memory address of the first machine instruction the compiler generates from Line 129. One needs a declaration for this array, using the macro jmp_buf (Line 60), which again is defined in one of the #include files.)

4.2 Message queues <sys/msg.h>

4.3 Semaphores

4.4 Shared Memory

4.5 Sockets

---

5. Miscellaneous Programming

5.1 Terminal I/O

5.2 System Information

5.3 Use of tools

---

6. Writing Larger Programs (Using

Makefiles)

When writing a large program, you may find it convenient to split it several source files. This has several advantages, but makes compilation more complicated.
Where the file contains the definition of an object, or functions which return values, there is a further restriction on calling these functions from another file. Unless functions in another file are told about the object or function definitions, they will be unable to compile them correctly. The best solution to this problem is to write a header file for each of the C files. This will have the same name as the C file, but ending in .h. The header file contains definitions of all the functions used in the C file. Whenever a function in another file calls a function from our C file, it can define the function by making a #include of the appropriate .h file.
Any file must have its data organised in a certain order. This will typically be:
A preamble consisting of #defined constants, #included header files and typedefs of important datatypes.
Declaration of global and external variables. Global variables may also be initialised here.
One or more functions.

C Style Guide

A good C Style guide is presented below:

File Names

- File names are not limited to 8.3
- File names are limited to Roman alphanumeric characters. Do not use underscores or spaces.
- File names should begin with a letter.
- File names shoud be mixed case. Every word (starting with the first word) has an initial capital character.
- Numbers indicating sequence must not be used. Numbers indicating functionality may be used.
- File names should describe the subsystem the file implements.
- A module with multiple files has file starting with the module name, then the subsystem name.
- Nouns are singular, not plural.
- Extensions: CSourceFile.c, CHeaderFile.h, CPlusPlusSourceFile.cpp, CPlusPlusHeaderFile.h, Library.lib, WindowsResource.rc, MacintoshResource.rsrc, and TextResource.r. Extensions for project files should be the development environment's default unless it contains non-alphanumeric characters.
- Only standard abbreviations may be used. Never use "2" and "4" for "To" and "For".

Header Files

- Tabs are 3 spaces.
- Maximum line length is 80 printable characters.
- Header files contain the specification for one subsystem.

- Header files should stand alone. I.E. every header file should include all other header files that it needs definitions from.
- Header files only contain declarations, never definitions.
- Header files internal to a module end with "Private".
- Header files are formatted as follows:

```
/***************************************************************************\
**
** <filename>.h
**
** <description>
**
** Copyright (c) <yyyy[-yyyy]> XYZ, Inc.
** All Rights Reserved
**
** <general comments>
**
** <project inclusion>
**
** $Header: $
**
\***************************************************************************/

#ifndef <filename>_H_
#define <filename>_H_

#ifndef <modulename>_PRIVATE
#error This file contains private data only.
#endif

/* Required Includes ********************************************************/

/* Constants ****************************************************************/

/* Forward Declarations *****************************************************/

/* Types ********************************************************************/

/* Extern Data **************************************************************/

/* Function Prototypes ******************************************************/

/* Inline Prototypes ********************************************************/

/* Macro Functions **********************************************************/

/* Class Declarations *******************************************************/

#endif // ifndef <filename>_H_

/***************************************************************************\
**
```

```
** $Log: $
**
\******************************************************************************/
```

Source Files

- Tabs are 3 spaces.
- Maximum line length is 80 printable characters.
- Source files contain the implementation for one subsystem of a project. "Dumping ground" files should not be used.
- The size of a file is an engineering decision.
- Source files are formatted as follows:

```
/******************************************************************************\
**
** <filename>.c
**
** <description>
**
** Copyright (c) <yyyy[-yyyy]> XYZ, Inc.
** All Rights Reserved
**
** <general comments>
**
** <project inclusion>
**
** $Header: $
**
\******************************************************************************/

/* Required Includes **********************************************************/

/* Constants ******************************************************************/

/* Types **********************************************************************/

/* Global Data ****************************************************************/

/* Static Data ****************************************************************/

/* Function Prototypes ********************************************************/

/* Functions ******************************************************************/

/******************************************************************************\
**
** $Log: $
**
\******************************************************************************/
```

Comments

- Comments document what or why, not how.
- Comments are indented with code.
- Every non-trivial local variable declarations should have a comment.
- Every field of a struct, union, or class should have a comment. These comments are aligned horizontally.
- Every file has a comment.
- Every function and method has a comment which is formatted as follows:

```
/**
 *
 * Function Name
 *
 * Description
 *
 * @param Parameter Direction Description
 * @param Parameter Direction Description
 *
 * @return Description
 *
 * @exception Description
 *
 */
```

  - The opening line contains the open comment ("/*") followed by one asterisk ("*").
  - The enclosed lines begin with a space and one asterisk (" *") which is followed by one space (" ") and the text of the comment.
  - Every function header must contain a function name block.
  - Every function header must contain a description block.
  - If the function takes arguments, the function header must at least one line for each argument in the form @param <Parameter> <Direction> <Description>.
  - The function header must contain at least one line containing the description of the return value in the form @return <Description>.
  - The function header must contain at least one line containing the description of any exceptions thrown by the function in the form @exception <Description>. If it is possible for the function to throw more than one kind of exception, they all should be enumerated here.
  - A function header may optionally contain additional tags providing more information about the function. See below.
  - The closing line contains a space (" ") the close comment ("*/").

- Do not "line draw" within a comment except for file and function comments.
- Single line comments may be either C++ ("//") or C ("/* */") comments.

- Multiple line comments should be C ("/* */") comments. They are formatted as follows:

```
/* your comment goes here,
** and is continued here
*/
```

- Triple bar comments are formatted as follows:

```
// ||| Name Date - Comment
/* ||| Name Date - Comment */
/* ||| Name Date - Comment
** (continued on multiple lines)
*/
```

- Triple dollar comments are formatted as follows:

```
// $$$ Name Date Deadline - Comment
/* $$$ Name Date Deadline - Comment */
/* $$$ Name Date Deadline - Comment
** (continued on multiple lines)
*/
```

- Code which is commented out must have a comment describing why it is commented out and what should be done with it when.

Identifier Name Formatting

- Identifier names are composed of American english words.
- Identifer names are composed of roman alphanumeric characters.
- Underscores are discouraged, but may be used to separate module information from the rest of the identifier.
- Identifier names are mixed case (except where noted otherwise). Every word (starting with the first or second word as noted below) has an initial capital character.
- Identifier names should be descriptive of the purpose or content of the identifier.
- Common abbreviatios may be used. Do not remove vowels to form an "abbreviation".
- Single character or other short identifier names are strongly discouraged.

Preprocessor

- - Non-parameterized preprocessor macros are in ALL CAPS.
- - Macros with side effects are in ALL CAPS.
- - Other macros may be mixed case.

Types

- - Mixed case with first word capitalized.
- - If decorated, should end with "Struct", "Rec", "Ptr", or "Hndl" as appropriate.

Identifiers

- - Mixed case with first word lower-case, and second word capitalized. Any decarators at the start of the identifier count as the first word.
- - Scope decoration must be prepended to the identifier for the following:
-   Application and file globals - "g"
-   Constants and enumerated types - "k"
-   Function and class statics - "s"
-   Class member variables - "m"
- - If you use indirection decorators, append "H" or "P" to the identifier name for Handles or Pointers, respectively.

Functions

- - Mixed case with first word capitalized.

Preprocessor

- "#" does not have a space after it.
- There is one space between the directive and its argument.
- Within code, all "#"s are aligned in the left column. Within a block of preprocessor directives, "#"s may be indented (with tabs) logically.

#define

- - Formatted as follows:

```
#define Symbol Value
#define Symbol Value \
    Continued value
```

- - All parameters are fully parenthesized.
- - Entire value is fully parenthesized.

#if / #elif / #else / #endif

- - Formatted as follows:

```
#if Condition1
#elif Condition2 // if Condition1
#else // if Condition1 elif Condition2
```

#endif // if Condition1 elif Condition2 else - If one part is commented, all parts must be commented.

- - All but inner-most nested structures must be commented.
- - All structures that span much code (10 lines) must be commented.
- - If enclosing functions, function header must be enclosed.
- - If enclosing functions, only enclose one function per structure (unless the entire file is being enclosed).
- - Parens, braces, etc... must always balance.

Declarations

- Declare only one variable per line.
- Pointers are declared with asterisk next to variable name, not type.
- Declaration blocks are separated from code by one blank line.
- Variables should be declared in the smallest scope in which they are used.
- Variables (except for globals) must be declared immediately following an open brace.
- Do not hide variables by using the same names in nested scopes.
- In functions, declare input-only parameters "const" whenever possible.
- Global declarations always start in leftmost column.

Enumerations

- - One constant per line, commented.
- - Never use Enumerated types.

Structures, Unions

- - One field per line, commented.
- - Whoever modifies a structure is responsible for maintaining or writing all compatibility code for old file formats, XTensions, etc...
- - Repeated nested structures must not be defined inside another structure.
- - The field names "unused", "reserved", and "foralignment" have special meanings.
- - All fields must be appropriately aligned (word or paragraph for 16- or 32-bit fields, respectively).

Functions

- - All non-static functions must have a prototype in a header file.
- - All prototypes must include a calling convention.
- - Continued line are broken after a comma, and indented two tabs.

Statements

- See Statement Formatting for more detail.

- Statement keywords are followed by a space.
- Single line comment bodies must be enclosed in braces.
- Open braces are placed at the end of the line (except for functions, where the brace goes on a line by itself).
- Close braces are placed on a line by themselves (except for "do {} while" and "if {} else if" statements).
- Continued lines are broken before an operator, and indented two tabs.
- Within a switch statement, cases are not indented.
- Return parameters are enclosed in parenthesis.
- Gotos should be used for jumping to a function clean up point.
- Lables should be "cleanup".

Expressions

- See Expression Formatting for more detail.
- The following operators do not have spaces between them and their operands:
- +(unary) -(unary) ++ -- ! ~ * & :: . -> () [] ,
- The following operators do have spaces between them and their operands:
- && || : = += /= -= *= == != < &gr; <= >= ?:
- The following operators should not have spaces between them and their operands:
- +(binary) -(binary) % / * ^ | & >> <<
- Parenthesis are encouraged to enhance clarity, even if they are not necessary. Continued lines are broken before an operator, and indented two tabs.

The prototype may occur among the global variables at the start of the source file. Alternatively it may be declared in a header file which is read in using a #include. It is important to remember that all C objects should be declared before use.

Compiling Multi-File Programs

This process is rather more involved than compiling a single file program. Imagine a program in three files prog.c, containing main(), func1.c and func2.c. The simplest method of compilation would be

```
gcc prog.c func1.c func2.c -o prog
```

If we wanted to call the runnable file prog we would have to type

We can also compile each C file separately using the cc -c option. This produces an object file with the same name, but ending in .o. After this, the object files are linked using the linker. This would require the four following commands for our current example.

```
gcc -c prog.c
gcc -c func1.c
gcc -c func2.c
gcc prog.o func1.o func2.o -o prog
```

Each file is compiled before the object files are linked to give a runnable file.

The **advantage of separate compilation** is that **only files which have been edited since the last compilation need to be re-compiled when re-building the program**. For very large programs this can save a lot of time. The **make utility** is designed to automate this re-building process. It checks the times of modification of files, and selectively re-compiles as required. It is an excellent tool for maintaining multi-file programs. Its use is recommended when building multi-file programs.

Make knows about `dependencies' in program building. For example;

We can get prog.o by running cc -c prog.c.
This need only be done if prog.c changed more recently than prog.o.
make is usually used with a configuration file called **Makefile** which describes the structure of the program. This includes the name of the runnable file, and the object files to be linked to create it. Here is a sample Makefile for our current example

```
#  Sample Makefile for prog
#
# prog is built from prog.c func1.c func2.c
#

# Object files (Ending in .o,
# these are compiled from .c files by make)
OBJS       = prog.o func1.o func2.o

# Prog is generated from the object files
prog: $(OBJS)
        $(CC) $(CFLAGS) -o prog $(OBJS)
```

When make is run, Makefile is searched for a list of dependencies. The compiler is involved to create .o files where needed. The link statement is then used to create the runnable file.**make** re-builds the whole program with a minimum of re-compilation, and ensures that all parts of the program are up to date.

The make utility is an intelligent program manager that maintains integrity of a collection of program modules, a collection of programs or a complete system -- does not have be programs in practice can be any system of files. In general only modules that have older object files than source files will be recompiled.
Make programming is fairly straightforward. Basically, we write a sequence of commands which describes how our program (or system of programs) can be constructed from source files.

The construction sequence is described in
makefiles which contain dependency rules and construction rules.

A dependency rule has two parts - a left and right side separated by a :

```
   left side : right side
```

The left side gives the names of a target(s) (the names of the program or system files) to be built, whilst the right side gives names of files on which the target depends (eg. source files, header files, data files). If the target is out of date with respect to the constituent parts, construction rules following the dependency rules are obeyed. So for a typical C program, when a make file is run the following tasks are performed:

1. The makefile is read. Makefile says which object and library files need to be linked and which header files and sources have to be compiled to create each object file.
2. Time and date of each object file are checked against source and header files it depends on. If any source, header file later than object file then files have been altered since last compilation THEREFORE recompile object file(s).
3. Once all object files have been checked the time and date of all object files are checked against executable files. If any later object files will be recompiled.

NOTE: Make files can obey any commands we type from command line. Therefore we can use makefiles to do more than just compile a system source module. For example, we could make backups of files, run programs if data files have been changed or clean up directories.

Creating a makefile

This is fairly simple: just create a text file using any text editor. The makefile just contains a list of file dependencies and commands needed to satisfy them.

Lets look at an example makefile:

```
prog: prog.o f1.o f2.o
  c89 prog.o f1.o f2.o -lm etc.

prog.o: header.h prog.c
   c89 -c prog.c

f1.o: header.h f1.c
   c89 -c f1.c



f2.o: ---
   ----
```

Make would interpret the file as follows:

1.
prog depends on 3 files: prog.o, f1.o and f2.o. If any of the object files have been changed since last compilation the files must be relinked.
2.
prog.o depends on 2 files. If these have been changed prog.o must be recompiled. Similarly for f1.o and f2.o.

The last 3 commands in the makefile are called explicit rules -- since the files in commands are listed by name.

We can use implicit rules in our makefile which let us generalise our rules and save typing.We can take

```
f1.o: f1.c
  cc -c f1.c

f2.o: f2.c
   cc -c f2.c
```

and generalise to this:

```
.c.o:   cc -c $<
```

We read this as .source_extension.target_extension: command

$< is shorthand for file name with .c extension.

We can put comments in a makefile by using the # symbol. All characters following # on line are ignored.

Make has many built in commands similar to or actual UNIX commands. Here are a few:

```
   break     date       mkdir

> type    chdir    mv (move or rename)
   cd    rm (remove)    ls
   cp (copy)    path
```

There are many more see manual pages for make (online and printed reference)

Make macros
We can define macros in make -- they are typically used to store source file names, object file names, compiler options and library links.
They are simple to define, e.g.:

```
SOURCES    = main.c f1.c f2.c
CFLAGS    = -g -C
LIBS    = -lm
PROGRAM    = main
OBJECTS    = (SOURCES: .c = .o)
```

where (SOURCES: .c = .o) makes .c extensions of SOURCES .o extensions.

To reference or invoke a macro in make do $(macro_name).e.g.:

$(PROGRAM) : $(OBJECTS)
$(LINK.C) -o $@ $(OBJECTS) $(LIBS)

NOTE:

$(PROGRAM) : $(OBJECTS) - makes a list of
dependencies and targets.
The use of an internal macros i.e. $@.
There are many internal macros (see manual pages) here a few common ones:

$*
-- file name part of current dependent (minus .suffix).
$@
-- full target name of current target.
$<
-- .c file of target.
An example makefile for the WriteMyString modular program discussed in the above is as follows:

```
#
# Makefile
#
SOURCES.c= main.c WriteMyString.c
INCLUDES=
CFLAGS=
SLIBS=
PROGRAM= main

OBJECTS= $(SOURCES.c:.c=.o)

.KEEP_STATE:

debug := CFLAGS= -g

all debug: $(PROGRAM)

$(PROGRAM): $(INCLUDES) $(OBJECTS)
 $(LINK.c) -o $@ $(OBJECTS) $(SLIBS)

clean:
 rm -f $(PROGRAM) $(OBJECTS)
```

Running Make
Simply type **make** from command line. UNIX automatically looks for a file called Makefile (note: capital M rest lower case letters). So if we have a file called Makefile and we type make from command line. The Makefile in our current directory will get executed. We can override this search for a file by typing make -f make_filename
e.g. **make -f my_make**
There are a few more -options for makefiles -- see manual pages.

7. Examples

8. Glossary Of Some Important Unix Commands & Concepts

Name Of Command/Concept	Syntax and Definition	Examples
grep	grep [options] PATTERN [FILE...] grep [options] [-e PATTERN | -f FILE] [FILE...] Grep searches the named input FILEs (or standard input if no files are named, or the file name - is given) for lines containing a match to the given PATTERN. By default, grep prints the matching lines.Patterns can be regexp patterns.	grep "pattern1" *.txt grep -v "pattern1" *.txt -- invert the sense of matching ie find those lines where the pattern is not found. grep -d recurse "pattern" *.cpp -- recursively searches for the pattern in all subdirectories.Alternatively, this can be written as grep -r "pattern" *.cpp. With -d other actions can be specified like "read" or "skip". In case of skip the directory is skipped. grep -c "pattern" *.txt -- gives the count of the matching lines suppressing the lines display. grep -i "pattern" *.* -- ignore case distinctions in both the pattern and the input files. grep -n "pattern" *.* -- prefix each o/p with line number within its input file. grep -w "pattern" *.* -- selects only those lines with complete word match. grep -x "some line" *.* -- select

		only those matches that exactly match the whole line.
fgrep/egrep	In addition, two variant programs egrep and fgrep are available. Egrep is the same as grep -E. Fgrep is the same as grep -F.	fgrep somePatternFile *.* -- obtain the patterns line by line from file. egrep "regexp pattern" *.*
at		
cron	Daemon to execute scheduled commands.	
tee		
shift		
who		
whoami		
export		
nice		
chmod		
umask		
ps		
cd		
ls		
vi		
sticky bit		
kill		
wc		
sed	sed [OPTION]... {script-only-if-no-other-script} [input-file]... -n, --quiet, --silent suppress automatic printing of pattern space -e script, --expression=script add the script to the commands to be executed	

	-f script-file, --file=script-file add the contents of script-file to the commands to be executed --help display this help and exit -V, --version output version information and exit If no -e, --expression, -f, or --file option is given, then the first non-option argument is taken as the sed script to interpret. All remaining arguments are names of input files; if no input files are specified, then the standard input is read.	
awk/gawk	awk [POSIX or GNU style options] -f progfile [--] file ... awk [POSIX or GNU style options] [--] 'program' file ... POSIX options: GNU long options: -f progfile --file=progfile -F fs --field-separator=fs -v var=val --assign=var=val -m[fr] val -W compat --compat -W copyleft --copyleft -W	

	copyright --copyright -W help --help -W lint --lint -W lint-old --lint-old -W posix --posix -W re-interval --re-interval -W traditional --traditional -W usage --usage -W version --version	
split		

Concept Name	Explanation in brief
Regular Expressions	A regular expression is a **pattern** that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. Grep understands two different versions of regular expression syntax: "**basic**" and "**extended**." In GNU grep, there is no difference in available functionality using either syntax. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards. The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash. A bracket expression is a list of characters enclosed by [and]. It matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit. Within a bracket expression, a range expression consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive, using the locale's collating sequence and character set. For example, in the default C locale, [a-d] is

equivalent to [abcd]. Many locales sort characters in dictionary order, and in these locales [a-d] is typically not equivalent to [abcd]; it might be equivalent to [aBbCcDd], for example. To obtain the traditional interpretation of bracket expressions, you can use the C locale by setting the LC_ALL environment variable to the value C.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their names are self explanatory, and they are [:alnum:], [:alpha:],[:cntrl:], [:digit:], [:graph:], [:lower:], [:print:],[:punct:], [:space:], [:upper:], and [:xdigit:]. For example, [[:alnum:]] means [0-9A-Za-z], except the latter form depends upon the C locale and the ASCII character encoding, whereas the former is independent of locale and character set. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.

The period . matches any single character. The symbol \w is a synonym for [[:alnum:]] and \W is a synonym for [^[:alnum]].

The caret ^ and the dollar sign $ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols \< and \> respectively match the empty string at the beginning and end of a word. The symbol \b matches the empty string at the edge of a word, and \B matches the empty string provided it's not at the edge of a word.

A regular expression may be followed by one of several **repetition operators:**

Repetition Operators	**Meaning**
.	Any Character.
?	The preceding item is optional and matched at most once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{n}	The preceding item is matched exactly n times.
{n,}	The preceding item is matched n or more times.
{n,m}	The preceding item is matched at least n times, but not more than m times.
[ ]	Delimit a set of characters. Ranges are specified as [x-y]. If the first character in the set is ^, then there is a match if the remaining characters in the set are not present.
^	Anchor the pattern to the beginning of the string. Only when first.
$	Anchor the pattern to the end of the string. Only when last.

WARNING

All these meta-characters have to be escaped with "\" if you want to use it in the search. For example, the correct string for looking for "e+" is: "e\+".

Examples	
^BA(B|F).*96	Select all nicknames that begin with BAB or BAF, and contain 96 elsewhere later.
.*	Match anything.
^[A-C]	Match anything that begin with letters though A to C.
^[^A-C]	Match anything that do not begin with letters through A to C.
^(BABA|LO)	Match anything that begin with BABA or LO.
C$	Match anything that end with C.
BABA	Match anything that contain, everywhere, BABA.

Two regular expressions may be concatenated; the resulting regular expression

	matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions. Two regular expressions may be joined by the **infix operator |**; the resulting regular expression matches any string matching either subexpression. Repetition takes precedence over concatenation, which inturn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules. The **backreference** \n, where n is a single digit, matches the substring previously matched by the nth parenthesized subexpression of the regular expression. In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \\\|, \(, and \).
Shell Scripting	UNIX provides a powerful command interpreter that understands over 200 commands and can also run UNIX and user-defined programs.
Pipe	where the output of one program can be made the input of another. This can done from command line or within a C program.
System Calls	UNIX has about 60 system calls that are at the heart of the operating system or the kernel of UNIX. The calls are actually written in C. All of them can be accessed from C programs. Basic I/0, system clock access are examples. The function open() is an example of a system call.

~The End~