Files

Whilst we are talking here about C on UNIX systems (e.g. Linux or Solaris), much but **not all** will be the same for other operating systems.

Files in C can be handled in two ways.

Firstly at the operating system level using (integer) file descriptors and the *open()*, *read()* and *write()* functions.

These are generally documented in section 2 of the manual since they are **system calls**.

Secondly, Files may be accessed as ***streams***.

Streams are higher level representations of files.

These functions (for example *fopen*, *fclose fread* and *fwrite*) are documented in section 3 of the unix manual.

File I-O at the stream level is managed by ***FILE*** structures.

These are managed by the C library functions, but we will need to declare pointers to a FILE structure if we wish to use files.

E.g.

```
#include <stdio.h>   /* needed for def. of FILE */

void main(void)
  {
     FILE *myfile;
                     /* declare a file pointer myfile */

     char *filename="fred.dat";
                     /* declare a string holding our
                        filename */


     myfile = fopen(filename,"w");
                    /* open the disk file fred.dat for
                       writing */

     if (myfile == NULL)
       {
          printf("Can't open %s for writing\n",filename);
          return;
       }
                   /* check for success */

     if (fclose(myfile) != 0)
       printf("Error closing %s\n",filename);
                  /* check it closed OK */
  }
```

Getting data out – putchar, putc, fputc, fputs;

As well as formatted output with *printf*, we also have standard functions to write strings and single characters to the screen or to a file.

On a UNIX system, the streams:

> stdin – standard input (generally your terminal or Xterm)
>
> stdout – standard output (ditto)
>
> stderr – standard error (generally the same place as stdout)

Are predefined for you.

In other words you can imagine your program has

```
FILE *stdin, *stdout, *stderr;
```

declared in in, with these opened and closed automatically.

```
putc('X',stdout);
```

&

```
putchar('X');
```

Therefore do exactly the same!

Lets have a look at the manual page (from Linux, generated as html using ***groff –man –Thtml***).

NAME

fputc, fputs, putc, putchar, puts - output of characters and strings

SYNOPSIS

#include <stdio.h>

int fputc(int *c*, **FILE** ***stream*);**
int fputs(const char ***s*, **FILE** ***stream*);**
int putc(int *c*, **FILE** ***stream*);**
int putchar(int *c*);
int puts(const char ***s*);

DESCRIPTION

fputc() writes the character *c*, cast to an **unsigned char**, to *stream*.

fputs() writes the string *s* to *stream*, without its trailing **'0'**.

putc() is equivalent to **fputc()** except that it may be implemented as a macro which evaluates *stream* more than once.

putchar(*c***);** is equivalent to **putc(***c***,***stdout***).**

puts() writes the string *s* and a trailing newline to *stdout.*

Calls to the functions described here can be mixed with each other and with calls to other output functions from the **stdio** library for the same output stream.

RETURN VALUES

fputc(), **putc()** and **putchar()** return the character written as an **unsigned char** cast to an **int** or **EOF** on error.

puts() and **fputs()** return a non negative number on success, or **EOF** on error.

CONFORMING TO

ANSI C, POSIX.1

BUGS

It is not advisable to mix calls to output functions from the **stdio** library with low level calls to **write()** for the file descriptor associated with the same output stream; the results will be undefined and very probably not what you want.

SEE ALSO

write(2), **fopen**(3), **fwrite**(3), **scanf**(3), **gets**(3), **fseek**(3), **ferror**(3)

Passing arguments into programs

How do we pass arguments to programs?

C has a simple but elegant approach which brings together some of the points already discussed.

main() is usually defined as:

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

main returns an int to whatever started the program, usually the shell.

If you are using a C-Shell (csh) you can obtain the value from status.

argc is a counter of the number of arguments (always at least 1)

****argv[]*** is an array of pointers to char, (or if you prefer a pointer to pointer to char).

Each element is a string, the first string being the program name, the rest being the program arguments.

Consider:

```
#include <stdio.h>

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

  for (i=argc; i > 0; i--)
    {
      fputs(argv[i-1],stdout);
      fputc('\n',stdout);
    }
  return 3;
}
```

If we compile this program with:

```
cc -o foo filename.c
```

And then type:

```
./foo fred joe bob
```

The output is:

```
bob
joe
fred
./foo
```

If in our shell (I'm assuming a C-shell here – ymmv)

```
echo $status
```

would print the value 3 (main's return value)

Error checking input

On most systems, the default for interactive I-O is buffered.

Processing input character by character and checking it is difficult.

For files however this is important!

Consider the program on the next page:

What are potential problems?
- What happens if I enter `fred`?
- What happens if I enter `9999999999999999999999999`?
- What happens if I enter `3.14159`?

How would you deal with them?

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

int getint(void);

void main(void)
{
   int i;

   i = getint();
   printf("you entered %d\n",i);

}


int getint(void)
{
  char buf[80];
  int tmp, i, c;

  i=0;
  c = getchar();
  while ((i < 80) && (c != '\n'))
    {
      if (isdigit(c))
        {
          buf[i] = (char) c;
          i++;
        }
      c = getchar();
    }
  buf[i] = '\0';
  tmp = atoi(buf);

  return tmp;
}
```

Converting strings

If we treat all input as simply a stream of bytes, testing, checking, and converting these as necessary, our program will be relatively safe.

Relatively since we still have to prevent buffer overflow: *remember the commandment! Where you type "foo" someone may type "supercaliflagalisticexpealadotious"*.

Whilst we can write our own functions to manipulate ASCII strings, C supplies some useful ones:

From `<stdlib.h>`

```
int atoi(const char *string);
```
converts strings to integers

```
double atof(const char *string);
```
converts strings to double

C also provides a range of string handling functions defined in `<string.h>`.

```
int strcmp(const char *a, const char *b);
```
compares a & b; if a>b returns >0, if a=b returns 0 if b>a returns <0

```
char *strcat(char *a, const char *b);
```
concatenates (joins together) b onto the end of a

string.h provides many other string manipulation functions as well.

Malloc

Closely related to pointers, malloc() and its relatives are vital for many real world problems.

Sometimes, we don't know when we are writing a program, how much memory we'll need.

- How much memory to hold a text file? Is it a 2 line email or Tolstoy's War and Peace?

- Sometimes our memory requirements will change.

- We might enlarge an image for example, or as we type our text gets longer.

We also have to remember that local variables are allocated from the stack, which is usually relatively small.

We also need to consider data structures such as linked lists, queues, and stacks.

In order to deal with this C provides (in *stdlib*) a collection of functions to allocate, release and resize raw "chunks" of memory.

Of these the important two functions for are:

```
void *malloc(size_t size)
```

```
void free(void *ptr)
```

(realloc() is also very useful!)

Malloc allocates a block of memory of size bytes (if it can) and returns a pointer to that memory.

If it fails (since there is not enough memory available) it returns a NULL pointer. (*remember the commandment about null pointers!*)

Free releases memory allocated by malloc for further use.

In many cases you do not need to worry about freeing memory too much since it will be automatically released when your program exits. However good practice is to release memory you no longer need.

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

void main(void)
{
  unsigned char *image;
  int size;

  size = 640 * 480;
  image = (unsigned char *) malloc(size);

  if (image != NULL)
    fprintf(stderr,"Memory allocated OK\n");
  else
    fprintf(stderr,"Memory allocation FAILED\n");

}
```

Type modifiers

Correctly the ISO/ANSI C standard subdivides these into storage class specifiers and type qualifiers.

All of these are implementation and architecture dependent! In order to use many of these sensibly you need to know a lot about the system you’re working on.

Currently, this is quite an esoteric aspect of C, you can almost certainly get away with avoiding these (*for now*! ☺)

Type qualifiers

- const

  advises the compiler the object is unchanging, therefore it could be placed in ROM, or use optimised.

- volatile

  The opposite of const! Important where we have memory mapped I-O (**NOT** the case with MARCO), and also where two programs communicate through shared memory. Imagine I have a program:

```
main()
  {
    unsigned char *p;

    p = 0x0000021d;
    *p = 1;
    while (*p)
      do_something()
  }
```

The compiler may well assume (incorrectly) that I mean do_something() forever! Using the volatile keyword advises the compiler that the value (*p) may change without my program explicitly altering it.

- auto

  The default for local variables, unitialised, usually stack allocated.

- register

  Compiler hint: I'm going to use this auto variable a lot! Where possible the variable will be stored in a register (very fast access).

  The negative side is & doesn't work on this storage class.

- static

  Static implies a degree of privacy and permanence. Static variables inside a function retain their value across function calls and are initialised to zero not usually allocated from the stack:

```
void foo(void)
  {
    static int count;

    printf("called %d times\n",count++);
  }
```

  Static variables outside of a function (i.e. global) are private to that source file, as are static functions.

- extern

  Tellsio the compiler the identified is *defined* elsewhere. (Extern is a *declaration*.)

- typedef

  Creates a new type! The ISO C standard includes it as a storage class specifier for technical reasons. <u>It does not allocate storage</u>!

Compilation, Linking, Making, & Compilation Units

So far we've taken a quite relaxed view of the issues here.

Consider the following files:

Main.c

```
#include "./myhead.h"

int main(int argc, char *argv[])
{
  hello();
  world();
}
```

Myhead.h

```
#ifndef _MYHEAD_H_

#define _MYHEAD_H_

void hello(void);
void world(void);

#endif
```

Hello.c

```
#include <stdio.h>

void hello(void)
{
  printf("HELLO");
}
```

World.c

```
#include <stdio.h>

void world(void)
{
  printf(" WORLD!\n");
}
```

Makefile

```
testprog : main.c myhead.h libmylib.a
	gcc -o testprog main.c -L. -lmylib

libmylib.a : hello.o world.o
	ar -r libmylib.a hello.o world.o

hello.o : hello.c
	gcc -c hello.c

world.o : world.c
	gcc -c world.c
```

These few little files introduce several concepts:

- A custom header file
- Multiple compilation units
- A custom library
- Make based compilation
- The use of the pre-processor

The Preprocessor

#define

#include

#if

#ifdef

#ifndef

#else

#elif

#endif

And much more!

Bitfields

Most low level C programmers employ a direct idiom for manipulating bits.

We've done this so far using unsigned chars. This has advantages and disadvantages.
A an alternative C supports the concept of bitfields:

```
struct
  {
     unsigned int fieldA  : 2;
                          : 2;
     unsigned int fieldB  : 1;
  } bits;
```

Declares a 2 bit and 1 bit variable in a structure called bits separated by 2 unused bits.

- Can't take the address (&)
- Can be signed/unsigned or plain int (best to type though!)

We can access these as, for example:

```
bits.fieldA = 3
```

Caveat Emptor! Bitfields are TOTALLY implementation dependent!

Unions

Unions are similar to variant records in Pascal.

They let us store and access variables of different sizes and types in the same memory space.

Unions are accessed in the same manner as structs.

Consider the following:

```
union my_union
  {
    int x[2];
    long double y;
   char z;
  };

union my_union u;

u.y = 3.14159l;

printf("%d : %d\n", u.x[0], u.x[1]);
```

All legal struct operations are legal on unions.

A union can only be initialised with a value of its first type.

The above code is legal, but probably meaningless!

Again, Implementation dependent!

Now would be a good point to discuss endian-ness!

Endian-ness

The term comes from "***Gulliver's travels***"

(Much like half a byte being a nibble, computer scientists have a strange sense of humour!)

80x86 family are "little endian"

68xxx family are "big endian"

Endianess is a problem since it determines the byte order of integer variables.

Consider the short int (16 bit value) 0x0100

- On an 80x86 processor this would have the value 1 decimal.
- On a 68xxx processor this would have the value 256

On a Pentium the least significant byte is stored first, so if we wrote in our program:

```
X = 0xff00;
```

00 FF would actually be stored in memory. You can explore this with the debugger.

Consider the following code:

```
#include <stdio.h>

void main()
{
    unsigned int x = 0x00000001;

    unsigned char *p;

    p = (unsigned char *) &x;

    *p ? big = 0 : big = 1;
}
```

Implementation Issues, Optimisation & Alignment

The last few weeks we've been discussing some esoteric odds & sods of C.

Many of these are extremely implementation dependant!

If you want to use C in anger, you need to know your compiler and target processor architecture.

Optimisation also means know your compiler/architecture.

Consider a busy wait:

```
for (i=0;i<4000;i++);
```

A clever compiler may decide that:

```
i=4000;
```

is a much quicker to execute substitute!

Alignment is another source of gotcha's!

In many situations compilers may choose to align variables on word boundaries to improve memory access. So:

char x[5];

could be stored as: ([J] = 1 byte of junk):

x[0] [J] [J] [J] x[1] [J] [J] [J] x[2] etc.

The compiler will accommodate this with pointer/array arithmetic, but code such as the endian-ness test could break!

Macros

So far we've been a little reticent about the full power of the pre-processor.

#define TRUE   1

We've met

The pre-processor also supports the use of Macro substitution

```
#define MAX(X,Y)      ((X) > (Y) ? (X) : (Y))
```

Here the values of X & Y are also substituted.

e.g.

```
int x;
int count;

x = MAX(count,7);
```

would become

```
x = ((count) > (7) ? (count) : (7));
```

This is great fun, and lovers of obfuscated C will use macros extensively!

But:

```
#define assign(a,b) a=(char)b

        assign(x,y>>8)
```

becomes

```
         x=(char)y>>8  /* which is always zero,
                          probably not what you
                                        want */
```

Lets have a look at a couple of other evil examples what do these do?

```
MAX(i++, j++);
```

Consider:

```
#define DOUBLE(X)2 * X
```

Good or Bad?

Consider:

```
z = 5;
```

```
z = DOUBLE(z+2);
```

What is the new value of z?

BE CAREFUL!

Moving on

We've left very little of the C language out. To become competent programmers however we need to:

- Really know our compiler!
- Know the available C libraries.
- Know the system calls provided by our O.S.
- Be familiar with common algorithms
- Be familiar with data structures

These provide our toolbox!

"To a small boy with a hammer, everything looks like a nail"

Unknown

"Data structures + Algorithms = Programs"

Niklaus Wirth

Algorithms are ways of doing something

Data structures are what we use to store and impose order on data

Elementary Data Structures

We've already met arrays, structures, unions!

- Inflexible
- Fixed Size

Consider image processing, directory listings etc.
Think back to malloc/pointers/structs.

Linked Lists

The simplest more advanced data structure.

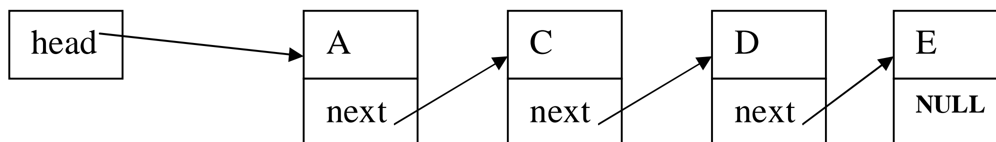
Imagine we want an easy to reorder, variably sized structure for holding letters.

```
struct node
  {
    char letter;
    struct node *next;
 };
```

We can then declare:

```
struct node *head, *nodepointer;
```

We can then build:



If we want to print out our list:

```
nodepointer = head;
while (nodepointer != NULL)
  {
    printf("%c\n",nodepointer->letter);
    nodepointer = nodepointer->next;
  }
```

Lets think about this structure a bit!

- How do we insert an item?
- How do we delete an item?
- How do we append an item?
- What sort of operations are harder or more time consuming than using an array?