C Programming - Lecture 3

- File handling in C - opening and closing.
- Reading from and writing to files.
- Special file streams stdin, stdout & stderr.
- How we SHOULD read input from the user.
- What are STRUCTURES?
- What is dynamic memory allocation?

File handling in C

- In C we use `FILE *` to represent a pointer to a file.
- `fopen` is used to open a file. It returns the special value `NULL` to indicate that it couldn't open the file.

```
FILE *fptr;
char filename[]= "file2.dat";
fptr= fopen (filename,"w");
if (fptr == NULL) {
    fprintf (stderr,
“ERROR”);
     /* DO SOMETHING */
}
```

Modes for opening files

- The second argument of `fopen` is the *mode* in which we open the file. There are three
  - "r" opens a file for reading
  - "w" opens a file for writing - and writes over all previous contents (deletes the file so be careful!)
  - "a" opens a file for appending - writing on the end of the file

The exit() function

- Sometimes error checking means we want an "emergency exit" from a program. We want it to stop dead.
- In main we can use "return" to stop.
- In functions we can use exit to do this.
- Exit is part of the `stdlib.h` library

```
exit(-1);
```

in a function is exactly the same as

```
return -1;
```

in the main routine

Writing to a file using fprintf

- fprintf works just like printf and sprintf except that its first argument is a file pointer.

```
FILE *fptr;
fptr= fopen ("file.dat","w");
/* Check it's open */
fprintf (fptr,"Hello World!\n");
```

- We could also read numbers from a file using fscanf – but there is a better way.

Reading from a file using fgets

- fgets is a better way to read from a file
- We can read into a string using fgets

```
FILE *fptr;
char line [1000];
/* Open file and check it is open */
while (fgets(line,1000,fptr) != NULL) {
   printf ("Read line %s\n",line);
}
```

fgets takes 3 arguments, a string, a maximum number of characters to read and a file pointer. It returns NULL if there is an error (such as EOF
Reads n-1 characters or up to a \n or EOF. It stores \n in the string.
It also adds \0 to the end of the string.
gets does not store \n.

Closing a file

- We can close a file simply using fclose and the file pointer. Here's a complete "hello files".

```
FILE *fptr;
char filename[]= "myfile.dat";
fptr= fopen (filename,"w");
if (fptr == NULL) {
    printf ("Cannot open file to write!\n");
    exit(-1);
}
fprintf (fptr,"Hello World of filing!\n");
fclose (fptr);
```

Common Error

- We use the file pointer to close the file - not the name of the file

```
FILE *fptr;
fptr= fopen ("myfile.dat","r");
/* Read from file */
fclose ("myfile.dat");
/* Ooops - that's wrong */
```

Three special streams

- Three special file streams are defined in the `stdio.h` header
- `stdin` reads input from the keyboard
- `stdout` send output to the screen
- `stderr` prints errors to an error device (usually also the screen)
- What might this do:

```
 fprintf (stdout,"Hello World!\n");
```

Reading loops

- It is quite common to want to read every line in a program. The best way to do this is a while loop using fgets.

```
/* define MAXLEN at start using enum */
FILE *fptr;
char tline[MAXLEN];  /* A line of text */
fptr= fopen ("sillyfile.txt","r");
/* check it's open */
while (fgets (tline, MAXLEN, fptr) != NULL) {
    printf ("%s",tline); // Print it
}
fclose (fptr);
```

Using fgets to read from the keyboard

- fgets and stdin can be combined to get a safe way to get a line of input from the user

```
#include <stdio.h>
int main()
{
    const int MAXLEN=1000;
    char readline[MAXLEN];
    fgets (readline,MAXLEN,stdin);
    printf ("You typed %s",readline);
    return 0;
}
```

Getting numbers from strings

- Once we've got a string with a number in it (either from a file or from the user typing) we can use `atoi` or `atof` to convert it to a number
- The functions are part of `stdlib.h`

```
char numberstring[]= "3.14";
int i;
double pi;
pi= atof (numberstring);
i= atoi ("12");
```

Both of these functions return 0 if they have a problem

Unintended Error

- fgets includes the '\n' on the end
- This can be a problem - for example if in the last example we got input from the user and tried to use it to write a file:

```
FILE *fptr;
char readfname[1000];
fgets (readfname,1000,stdin);
fptr= fopen (readfname,"w");
/* oopsie - file name also has \n */
```

Even experienced programmers can make this error

Structures in C

- In C, we can create our own data types - FILE is an example of this.
- If programmers do a good job of this, the end user doesn't even have to know what is in the data type.
- `struct` is used to describe a new data type.
- `typedef` is used to associate a name with it.
- `int`, `double` and `char` are types of variables. With `struct` you can create your own. It is a new way to extend the C programming language.

The struct statement

- Here is an example struct statement.

```
#include <stdio.h>
struct line {
    int x1, y1;  /* co-ords of 1 end of line*/
    int x2, y2;  /* co-ords of other end */
    };
main()
{
struct line line1;
.
.
}
```

This defines the variable line1 to be a variable of type line

Typedef

- Typedef allows us to associate a name with a structure (or other data type).
- Put typedef at the start of your program.

```
typedef struct line {
    int x1, y1;
    int x2, y2;
    } LINE;

int main()
{
LINE line1;

}
```

line1 is now a structure of line type
This is what was happening with all that FILE * stuff

Accessing parts of a struct

- To access part of a structure we use the dot notation

```
LINE line1;
line1.x1= 3;
line1.y1= 5;
line1.x2= 7;

if (line1.y2 == 3) {
    printf ("Y co-ord of end is 3\n");
    }
```

What else can we do with structs?

- We can pass and return structs from functions. (But make sure the function prototype is _AFTER_ the struct is declared)

```
typedef struct line {
    int x1,y1;
    int x2,y2;
} LINE;

LINE find_perp_line (LINE);
/* Function takes a line and returns a
perpendicular line */
```

What's the point of all this with structs?

- It makes your programming easier and it makes it easier for other programmers.
- FILE * was a typedef'd struct but you could use it without knowing what was inside it.
- You can extend the language - for example, if you use complex numbers a lot then you can write functions which deal with complex nos.

Complex no functions

```
typedef struct complex {
   float imag;
   float real;
} CPLX;

CPLX mult_complex (CPLX, CPLX);

int main()
{
/* Main goes here */
}

CPLX mult_complex (CPLX no1, CPLX no2)
{
    CPLX answer;
    answer.real= no1.real*no2.real - no1.imag*no2.imag;
    answer.imag= no1.imag*no2.real + no1.real*no2.imag;
    return answer;
}
```

Dynamic memory allocation

- How would we code the "sieve of Eratosthenes" to print all primes between 1 and n where the user chooses n?
- We _could_ simply define a HUGE array of chars for the sieve - but this is wasteful and how big should HUGE be?
- The key is to have a variable size array.
- In C we do this with DYNAMIC MEMORY ALLOCATION

A dynamically allocated sieve

```
#include <stdlib.h>
void vary_sieve (int);
void vary_sieve (int n)
/* Sieve to find all primes from 1 - n */
{
    char *sieve;
    int i;
    sieve= (char *)malloc (n*sizeof(char));
    /* check memory here */
    for (i= 0; i < n; i++)
        sieve[i]= UNCROSSED;
    /* Rest of sieve code */
    free (sieve);  /* free memory */
}
```

A closer look at malloc

- Let's look at that malloc statement again:

```
sieve= (char *)malloc(n*sizeof(char));
```

This is a CAST (remember them) that forces the variable to the right type (not needed)

we want n chars

sizeof(char) returns how much memory a char takes

This says in effect "grab me enough memory for 'n' chars"

Free

- The free statement says to the computer "you may have the memory back again"

```
    free(sieve);
```

essentially, this tells the machine that the memory we grabbed for sieve is no longer needed and can be used for other things again. It is _VITAL_ to remember to free every bit of memory you malloc

Check the memory from malloc

- Like fopen, malloc returns NULL if it has a problem
- Like fopen, we should always check if malloc manages to get the memory.

```
float *farray;
/* Try to allocate memory for 1000 floats */
farray= malloc(1000*sizeof(float));
if (farray == NULL) {
    fprintf (stderr, "Out of memory!\n");
    exit (-1);
    }
```

Pointers to struct example

```
typedef struct great_mathematician {
    char name[80];
    int year_of_birth;
    int year_of_death;
    char nationality[80];
} MATHEMATICIAN;
.
.
MATHEMATICIAN *cantor;
cantor= (MATHEMATICIAN *)malloc (sizeof (MATHEMATICIAN));
/* Check the allocation here */
(*cantor).year_of_birth= 1845;
(*cantor).year_of_death= 1918;
/* Remember to close comments with a diagonal slash */
strcpy ((*cantor).name, "Georg Cantor");
strcpy ((*cantor).nationality, "German");
free(cantor); /* Don't forget to free the memory */
```

Consider the peculiar syntax here

Pointers to struct (2)

- C allows `name->bit` as a shorthand for `(*name).bit`

```
MATHEMATICIAN *cauchy;
cauchy= (MATHEMATICIAN *)
   malloc (sizeof (MATHEMATICIAN));
        /* The sizeof Cauchy was quite large */
cauchy->year_of_birth= 1789;
cauchy->year_of_death= 1857;
strcpy (cauchy->name, "Augustin Louis Cauchy");
strcpy (cauchy->nationality, "French");
.
.
.
free(cauchy);
```

This shorthand is always used by C programmers

Passing it to a function

```
int main()
{
    MATHEMATICIAN *turing;
    turing= (MATHEMATICIAN *)
      malloc(sizeof(MATHEMATICIAN));
    set_up_turing(turing);
    /* Do stuff with the variable */
    free(turing);
}

void set_up_turing (MATHEMATICIAN *turing)
{
    turing->year_of_birth=1912;
    turing->year_of_death=1954;
       /* In tragic circumstances */
    strcpy (turing->name,"Alan Mathison Turing");
    strcpy (turing->nationality,"British");
}
```

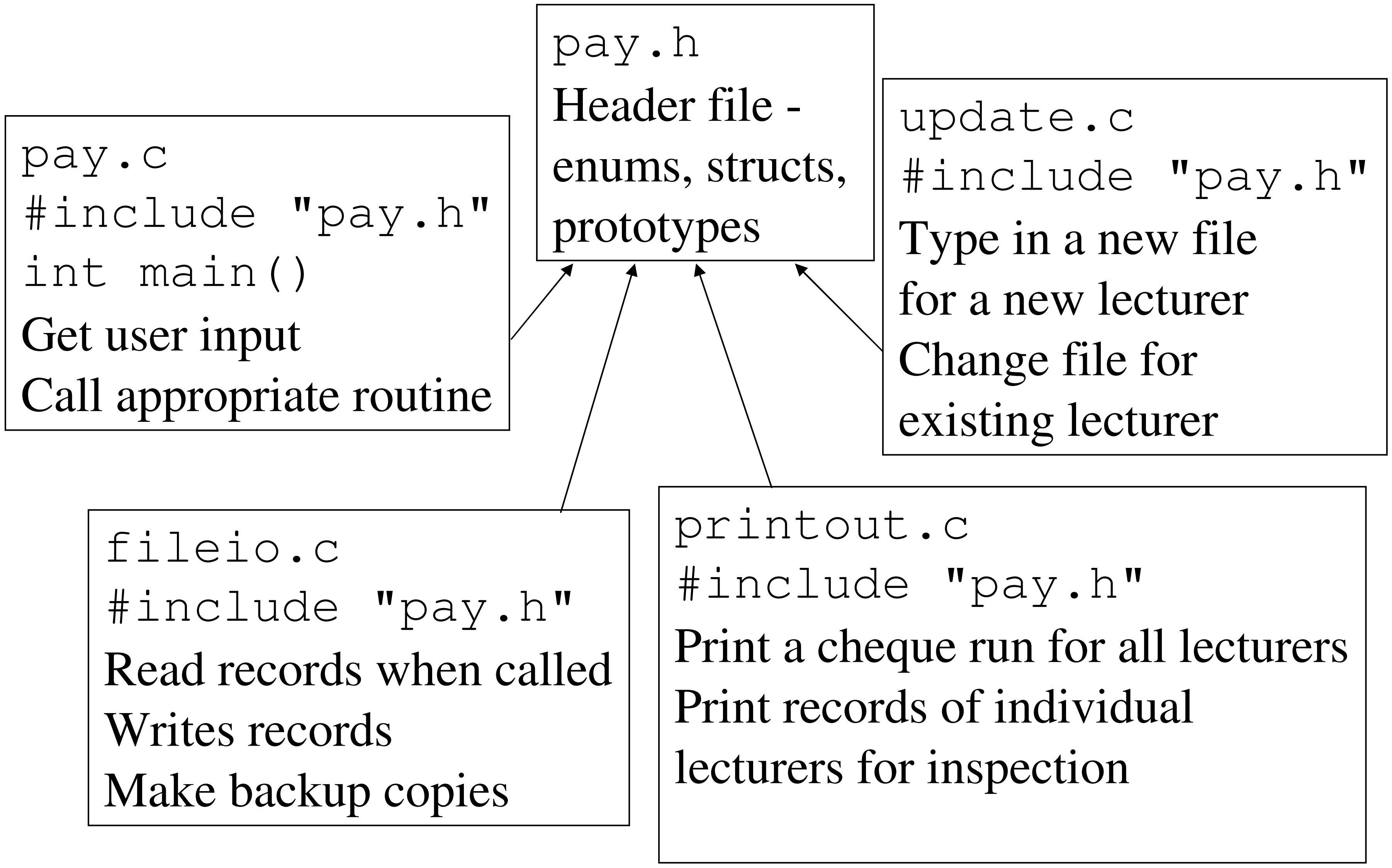
Multiple file programming - why do it?

- It means that more than one person can work on a program at once
- It means that when your program gets big, you don't have to scout through lots of functions
- It means that when you change 1 file of your huge program, you don't have to change it all
- (But on the other hand, it is more complicated)

Multi-file programming - how

- Create a "header file" (a file ending in .h) which contains all the prototypes, enums, #defines, structs, typedefs etc for your code
- Create a number of C source code files - they should all #include your header file.
- Source files NOT header files contain the functions.
- One (and only one) of your C source code files contains main.

Example structure



Things to note

- If a bit of source uses a function, it must have access to its prototype
- This is why prototypes are in the header
- Similarly for enums, #defines and struct definitions.
- Forgetting to #include the header file can seriously cause problems
- The functions go in the .c file not the .h file.
- It is traditional to use "" instead of <> to indicate a header file you wrote yourself

```
#include <stdio.h>    #include "myprog.h"
```

The extern statement

- If we want to use global variables, in multi-file programming?
- If we define them in the header then there will be multiple copies
- If we define them in one file, how will all files see them?
- The key is to define them in one file and declare them as "extern" in other files

Extern statement

- Only used with global variables in multi-file projects.
- Says to compiler "don't worry, this is dealt with elsewhere"

```
file1.c

int glob_array[100];
```

```
file2.c

extern int glob_array[100];
```

make and Makefile

Suppose you have three file, to compile these files using cc, the command would be

CC -o prog1 file1.cc file2.cc file3.cc

However, this is long and troublesome when typing many times.

Solution: Using make, a utility to buid object files according to the dependency rules defined in a Makefile.

Just type make in the command line will do the same work.

make and Makefile

Make is invoked from a command line with the following format

```
make [-f makefile] [-bBdeiknpqrsSt] [macro name=value] [names]
```

However from this vast array of possible options only the -f makefile and the names options are used frequently. The table below shows the results of executing make with these options.

Options	Result
make	use the default Makefile, build the first target in it
make myprog	use the default Makefile, build the target myprog
make -f mymakefile	use mymakefile as the Makefile, build the first target in it
make -f mymakefile myprog	use mymakefile as the Makefile, build the target

A simple Example of a Makefile

```
prog1 : file1.o file2.o file3.o
   CC -o prog1 file1.o file2.o file3.o
file1.o : file1.cc mydefs.h
   CC -c file1.cc
file2.o : file2.cc mydefs.h
   CC -c file2.cc
file3.o : file3.cc
   CC -c file3.cc
clean :
   rm file1.o file2.o file3.o
```

About Makefile

- Comments: begins with a pound sign (#). If multiple lines are needed each line must begin with the pound sign, i.e.
  # This is a comment line
- Dependency rules: A rule consist of three parts, one or more targets, zero or more dependencies, and zero or more commands in the following form:

```
    target1 [target2 ...] :[:] [dependency1 ...] [; commands]
            [<tab> command]
```

  Note: each command line must begin with a tab as the first character on the line and only command lines may begin with a tab.
- Target: A target is usually the name of the file that make creates, often an object file or executable program. The target name must show up in the first column of the Makefile.
- Phony target: A phony target isn’t really the name of a file. It will only have a list of commands and no prerequisites.

About Makefile

- Commands: Each command in a rule is interpreted by a shell to be executed.
- Macros: for defining constants, thus avoiding repeating text entries and make descriptor files easier to modify. Macro definitions have the form

```
NAME1 = text string
NAME2 = another string
```

Macros are referred to by placing the name in either parentheses or curly braces and preceding it with a dollar sign ($). For example

```
$(NAME1)
${NAME2}
```

- Some commonly-used macro names

```
LIBS = -lm -lglut -lGL              #libraries
OBJS = file1.o file2.o file3.o      #object files
CXX = CC                        #compiler
DEBUG_FLAG = -g                     # assign -g for debugging
SHELL = /bin/sh                     # the standard shell for make
```