C Programming – Chapter 5

- In this lecture we learn about:
  - Good programming practices
  - How to document your code
  - How to test your code
  - How to design your code

Types of documentation

- Internal documentation (comments in your code)
- External programmer documentation (for other programmers who would work with your code)
- User documentation (the manual for the poor fools who will be using your code)

How to write good comments

- Does your comment help your reader understand the code?
- Are you writing a comment just because you know that "comments are good"?
- Is the comment something that the reader could easily work out for themselves?

Some common bad comments

```
i= i+1;  /* Add one to i */

for (i= 0; i < 1000; i++) { /* Tricky bit */
.
. Hundreds of lines of obscure uncommented code here
.
}
int x,y,q3,z4;  /* Define some variables */

int main()
/* Main routine */

while (i < 7) { /*This comment carries on and on
```

How comments can make code worse

```
while (j < ARRAYLEN) {
    printf ("J is %d\n", j);
    for (i= 0; i < MAXLEN; i++) {
        for (k= 0; k < KPOS; k++) {
            printf ("%d %d\n",i,k);
        }
    }
    j++;
}
```

Some more bad comments

```
while (j < ARRAYLEN) {
    printf ("J is %d\n", j);
    for (i= 0; i < MAXLEN; i++) {
/* These comments only */
        for (k= 0; k < KPOS; k++) {
/* Serve to break up */
            printf ("%d %d\n",i,k);
/* the program */
        }
/* And make the indentation */
    }
/* Very hard for the programmer to see */
    j++;
}
```

How much to comment?

- Just because comments are good doesn't mean that you should comment every line.
- Too many comments make your code hard to read.
- Too few comments make your code hard to understand.
- Comment only where you couldn't trivially understand what was going on by looking at the code for a minute or so.

What should I always comment

- Every file (if you do multi-file programming) to say what it contains
- Every function – what variables does it take and what does it return. (I like to comment the prototypes too slightly to give a hint)
- Every variable apart from "obvious" ones (`i,j,k` for loops and `FILE *fptr` don't require a comment but `int total;` might)
- Every struct/typedef (unless it's really trivial)
- Every block of code that doing something specific

Other rules for comments

- Comment if you do something "weird" that might fool other programmers.
- If a comment is getting long consider referring to other text instead
- Don't let comments interfere with how the code looks (e.g. make indentation hard to find)

External (programmer) documentation

- This tells other programmers what your code does
- Most large companies have their own standards for doing this
- The aim is to allow another programmer to use and modify your code without having to read and understand every line
- This is just ONE way of doing it – everyone has their own rules.

External documentation (Stage 1)

- Describe how your code works generally
- What is it supposed to do?
- What files does it read from or write to?
- What does it assume about the input?
- What algorithms does it use

External Documentation (stage 2)

- Describe the general flow of your program (no real need for a flowchart though)
- Diagrams can help here.
- Explain any complex algorithms which your program uses or refer to explanations elsewhere. (e.g. "I use the vcomplexsort see Knuth page 45 for more details")

External documentation(stage 3)

- If you use multi-file programming explain what each file contains
- Explain any struct which is used a lot in your program
- You might also like to explain (and justify) any global variables you have chosen to use

External documentation (stage 4)

- Describe every "major" function in your program
- Describe what arguments must be passed and what is returned.
- (It is up to you to decide what is a "major" function – and really depends on the level of detail you wish to document to).
- Consider which functions are doing "the real work" – they might not necessarily be the longest or most difficult to write ones.

User documentation

- This is documentation for the user of your program
- It is the "user manual"
- Entire books have been written on the subject and I don't intend to cover it here
- Feel free to include user documentation for your project if you want (but not too much of it)

Error checking in programs

- A good programmer always checks file reading, user input and memory allocation (see later) for errors.
- Nothing convinces a user of your program that you're an idiot faster than it crashing when they type "the wrong thing".
- Take some action to avert the error even if that action is stopping the program.
- It is best to print errors to the stderr stream.

```
fprintf (stderr,"There is an error!\n");
```

What should I check and what should I do?

- Check any operation that could fail. Check every time a file is opened or memory is allocated.
- Check user input unless there is no possible way this could break the program.
- In the case of out of memory it is usually best just to print an error exit(-1);
- In the case of user input then give them a second chance.
- In the case of file names it may be a user input problem.

The classic user input errors

- Could a user input cause something to run off the end of an array or overflow a variable (too big)?
- Is it a problem if the user input is negative or very tiny? (too small).
- Is there a possibility of a divide by zero? (exactly too medium sized).
- In 1998 the guided-missile carrier USS Yorktown was shut down for several hours when a crew-member mistakenly input zero to one of the computers.

Wrappered function

- Isn't it pretty boring to write an error check every time you try a malloc?
- Most of the time, after all, you just say "out of memory" and exit.
- It seems like lots of effort to write the same bit of code every time.
- The solution is to write your own "wrappered" malloc.
- You might want to "wrapper" other functions (for example file open).

safe_malloc

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

void *safe_malloc (size_t);
/* Error trapping malloc wrapper */

void *safe_malloc (size_t size)
/* Allocate memory or print an error and exit */
{
    void *ptr;
    ptr= malloc(size);
    if (ptr == NULL) {
        fprintf (stderr, "No memory line:%d file:%s\n",
      __LINE__, __FILE__);
        exit(-1);
    }
    return ptr;
}
```

Good programming practice "a clean interface"

- Interface in this sense means the functions you provide and how they are accessed.
- A good programmer makes their code useful to other programmers.
- The best way to do this is to write reusable functions which are easy to understand.
- If your functions are good then there shouldn't be _too_ many arguments passed to them.
- Think about "what do I need to pass to this function" and "what do I need back from it".
- If you have written a clean interface then your functions will almost explain themselves.

By using structs, we can make our functions look simpler

- Sometimes we need to pass a LOT of information to a function.

```
void write_record (FILE *fptr, char name[], int wage,
   int hire_date, int increment_date, int pay_scale,
   char address[])
/* Function to write info about a lecturer */
{
```

Nicer to bundle it as a struct - and we can add stuff later

```
}
void write_record (FILE *fptr, LECTURER this_lecturer)
/* Function to write info about a lecturer */
{

}
```

Functions should be "consistent"

- If you write a lot of similar functions it is best to make them work the "same way"

```
int write_record (char fname[],LECTURER lect)
/* Return -1 for FAIL 0 for success */
{

}
```

The second function is perverse given the first

```
int add_record (LECTURER lect, char fname[])
/* Return 0 for FAIL 1 for success */
{

}
```

Another programmer reading your code will
be justified in anything short of actual bodily harm
if your code works like this.

Functions should be predictable

- Don't make your function change arguments it doesn't NEED TO. Unless your function is explicitly supposed to change arrays, DON'T change the array.

```
FILE *fptr;
char fname[]= "file.txt";
fptr= fopen (fname, "r");
if (fptr == NULL) {
    printf ("Can't open %s\n",fname);
    return -1;
    }
```

Wouldn't you be annoyed if fopen had unexpectedly changed what was in fname?

Buffer Overflow

- What happens to this code if it is given a longer string in str2 than str1?

```
void strcpy(char str1[], char str2[])
/* Copy to str1 from str2 */
{
    int i= 0;
    while ((str1[i]= str2[i]) != '\0')
        i++;
}
```

Another complex line which assigns and compares.

Where do buffer overflows come from?

- Here are just some common ways that buffer overflows might arise
  - Incautious use of "strcpy" (copying a potentially larger string into a smaller one).
  - Use of the gets command instead of fgets from stdin (which is why I didn't even teach you about gets).
  - Forgetting to check array bounds on input strings.

Testing your code really works

- Just because a piece of code works once doesn't mean it will work again.
- Because we get a right answer for an input of 'n' does not mean we will get the right answer for 'm'.
- Working code should never "crash" it should always exit with an error whatever its input.
- You should know how your code will behave when asked "the wrong question".

Test boundary conditions

- Consider what might happen if the input is very large or very small.
- If there is a possibility that your code will get such input you should make sure it can deal with it.
- Always beware of the divide by zero error.
- In 1998 the guided-missile carrier USS Yorktown was shut down for several hours when a crew-member mistakenly input zero to one of the computers. Don't let your code work like this.

Boundary conditions example

- What is wrong with this code which is supposed to be like strlen

```
int my_strlen (char *string)
/* What is wrong with this code to find the length
of a string */
{
    int len= 1;
    while (string[len] != '\0')
        len++;
    return len;
}
```

Overflows of numbers

- If you are going to work with large numbers then be sure you know how large a number your variables can deal with.
- In most implementations of C an unsigned char can be from 0 to 255. How big an int can be varies from computer to computer.
- In July 1996 Ariane 5 exploded as a direct result of a programming error which tried to fit 64 bits of floating point into a 16 bit int.

What if the user asks "the wrong question"

- This code finds the average of 'n' doubles - under what conditions does it fail.

```
double avg (double a[], int n)
/* a is an array of n doubles */
{
    int i;
    double sum= 0;
    for (i= 0; i < n; i++) {
        sum+= a[i];
    }
    return sum/n;
}
```

Program defensively

- In some cases (not all) you might add code to weed out rogue values.

```
void class_of_degree (char degree[], double percent)
/* Work out the approx. class of degree from
someone's percentage overall mark */
{
    if (percent < 0 || percent > 100)
        strcpy(degree,"Error in mark");
    else if (percent >= 70)
        strcpy(degree,"First");
    else if (percent >= 60)
        strcpy(degree,"Two-one");
    .
    .
}
```

These lines are just here out of caution

How to test your code while writing

- A good programmer doesn't sit down, write 10,000 lines of code and then run it.
- It will make your life easier if you test your program as you write it.
- Write the smallest possible part of the program you think will _do something_ and test it.
- Build the program up gradually - testing as you go.
- I like to compile every dozen lines or so – as soon as I've made a significant change. (I use a separate window to compile in).

When and what to test

- If your program takes no input but simply runs and produces an answer then it may not need much testing. Most programs are not like this.
- If you are doing the cryptography project or Zipf's law projects, for example, your programs should be taking strings of input.
- What would happen if those strings of input were just rubbish instead of well behaved strings of words and letters.

Document your testing

- Documenting your testing is critical and it will be important in your project.
- If appropriate, you should include in your write up, some evidence that you have tested your code with various inputs
- Failing to document testing can have important consequences
- One of the problems which beset the Pathfinder probe had actually been spotted in testing before launch - but forgotten about. It had to be solved while in flight.

Document your testing

- Documenting your testing is critical and it will be important in your project.
- If appropriate, you should include in your write up, some evidence that you have tested your code with various inputs
- Failing to document testing can have important consequences
- One of the problems which beset the Pathfinder probe had actually been spotted in testing before launch - but forgotten about. It had to be solved while in flight.