# COMP281
# Principles of C and memory management

## Lecture 10

**Dr. Frans Oliehoek**
**Department of Computer Science**
**University of Liverpool**

## Last Time

- Wrap up: dynamic data structures and ADTs
  - grow memory as needed
  - abstract data types that may be useful to implement
- Function pointers provide a method of storing some function to call
  - Can be stored in arrays and accessed by index
  - Can be used for ‘callbacks’, for use with standard library functions

## Today

- Programs over multiple files
  - compilation of complex projects
- File Handling
- Overview to get ready to tackle the last assignment!
  - Reminder about debugging
  - Common Mistakes
  - Questions
- A few useful pointers as appendix to these slides

## About Assignment 3

## Assignment 3

- Show that you can use dynamic data structures
- Generic implementations are better
- Demonstrate understanding of casting
- Don't use too much memory, avoid leaks

## Programs over Multiple Files & Compilation of Complex Projects

## Header files

- Common to put function prototypes into a header file (usually with the same name as the file)
  - easily separate your program into multiple files

**maths.h:**

```
int multiply(int a, int b);
```

**maths.c:**

```
int multiply(int a, int b)
{
    return a*b;
}
```

**main.c:**

```
#include <stdio.h>
#include “maths.h”
main()
{
    int product = multiply(10,20);
    printf(“%d \n”,product);
}
```

# Header files

- Common to put function prototypes into a header file (usually with the same name as the file)
  - easily separate your program into multiple files

**maths.h:**

```
int multiply(int a, int b);
```

**maths.c:**

```
int multiply(int a, int b)
{
    return a*b;
}
```

**main.c:**

```
#include <stdio.h>
#include “maths.h”
main()
{
    int product = multiply(10,20);
    printf(“%d \n”,product);
}
```

Visibility of functions:

- function **declaration** from other files (math.h) can be seen in including file (main.c)
- function **definition** is assumed to be done in some other file
- i.e., the function definition is 'external' to the file main.c

# The `external` keyword

- Can define variables as being 'external' to the file
  - e.g., `extern int train_number;`
- This essentially is the "declare, but do not define" for global variables.
  - "define" means: reserve memory!
- Hence a header file *can* contain all of the functions and variables that may be accessed from another file.
  - i.e., everything that might be declared 'public' in Java
  - some tools may create header files automatically.
  - (not necessary to have it there, might still access otherwise...)
- #include is a simple preprocessor directive
  - it just copies the file given into the source
  - often only used for headers
  - but could have other uses

**maths.h:**

```
//declaration
external int number;
```

**maths.c:**

```
//definition
int number = 42;
```

**main.c:**

```
#include <stdio.h>
#include "maths.h"
main()
{
    int product =
      multiply(10,number);
    printf("%d \n",product);
}
```

external_math.c
external_math.h
external_main.c

## Compiling multiple files

- Approach 1: just specify all the files….

```
$ gcc main.c maths.c -o approach1
```

$$\begin{tikzcd}\text{main.c, maths.c} \arrow[r] & \text{approach1}\end{tikzcd}$$

- A large project may contain thousands of source files
  - can take a **long** time to compile each file every time (hours…!)
- Solution: compile files separately
  - only re-compile changed .c files

creates an 'object' files: compiled, but contains only symbolic links to external variables and functions

'links' the object files: symbolic links are removed and replaced with the correct addresses

```
$ gcc -c main.c -o main.o
$ gcc -c maths.c -o maths.o
$ gcc main.o maths.o -o approach2
```

$$\begin{tikzcd}\text{main.c} \arrow[r] & \text{main.o} \arrow[dr] & \\ & & \text{approach2} \\ \text{maths.c} \arrow[r] & \text{maths.o} \arrow[ur] & \end{tikzcd}$$

## Linking Multiple Files

- There are two stages to compiling a program
  - Compiling: turning the C source into machine-readable instructions
  - Linking: resolve symbolic links into addresses

You may need to tell the compiler (and linker, if it is separate) how to link to other libraries

  - Libraries contain code that has been compiled
  - The function calls need to be linked to your program

# Makefiles

- For a large project, it is common to create a Makefile,
  - knows how to build the entire project
- To run, simply type `make <TARGET>`

```
CC = gcc
CFLAGS  =  -Wall -O3
main:  main.o maths.o
   $(CC) $(CFLAGS) -o main main.o maths.o

main.o:  main.c maths.h
   $(CC) $(CFLAGS) -c main.c

maths.o:  maths.c
   $(CC) $(CFLAGS) -c maths.c

clean:
   rm main *.o
```

to make main, it needs main.o and maths.o

how to make main.o, if either main.c or maths.h has been changed

# Makefiles

Another example with some more features:

```
SRCS = main.c maths.c
CC = gcc
CFLAGS = -Wall -O3
OBJS = $(SRCS:.c=.o)
MAIN = main

$(MAIN): $(OBJS)
   $(CC) $(CFLAGS) -o $(MAIN) $(OBJS)

%.o:%.c
   $(CC) $(CFLAGS) -c $< -o $@

clean:
   rm *.o $(MAIN)
```

to make main, it needs main.o and maths.o

rule that explains how to transform **any** .c file to an .o file

# Compilation parameters

- We have seen how to conditionally compile code (with #IF)
  - Defines can be set on the command line with the –D option
  - e.g., -Wall -DDEBUG_LEVEL=2
  - -Wall -O2 -DFINAL -DDEBUG_LEVEL=2  (the default sets FINAL = 1)

```
#if DEBUG_LEVEL > 1
printf(“entering critical section”);
#endinf
```

- We can also check to see if something was defined (or not)

```
#ifdef DEBUG_LEVEL
printf(“entering critical section”);
#endinf

#ifndef DEBUG_LEVEL
  #define DEBUG_LEVEL 0
#endif
```

- This is often used in case a header file is included twice…

## Avoiding multiple inclusion

- A header file may include other (required) header files, which may include more files...
  - could end up including the same file .h multiple times

**external_math.h:**

```
#ifndef  EXTERNAL_MATH_INC
#define  EXTERNAL_MATH_INC

int multiply(int a, int b);

extern int int_defined_by_external_math_c;

#endif   /* -- #ifndef EXTERNAL_MATH_INC  -- */
```

# Compiler Optimization

- http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- The compiler can use a variety of techniques to improve the efficiency of your code.
  - "-O0" no optimization; debug mode
  - ...
  - "-O3" many optimizations done

# Compiler Optimization vs Debugging

- Optimized code can be harder to use with a debugger
  - program instructions may not be what you would expect
  - they may even be in the ‘wrong’ order
  - variables and code may even be removed entirely
- Common: ‘debug’ and ‘release’ versions
  - with the same code, but generating two possible output program

| Debug | Release |
|---|---|
| No code optimization | All required optimizations |
| May print debug information | Only prints fatal errors |
| May store a logfile of all events | Only logs important information |
| May check for conditions that could potentially cause errors | Only checks for critical errors |

## File Handling

## File Handling

- Similarly to Java, files are accessed with a FILE handle
  - the file handle is created by the system, you access it via a pointer
  - `FILE *fopen(const char *path, const char *mode);`
  - e.g., `FILE *main_file = fopen(“data.txt”, “r”);`

| | |
|---|---|
| “r” | **read:** Open file for input operations. (The file must exist.) |
| "w" | **write:** Create an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file. |
| "a" | **append:** Open file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist. |
| "r+" | **read/update:** Open a file for update (both for input and output). The file must exist. |
| "w+" | **write/update:** Create an empty file and open it for update (both for input and output). If a file with the same name already exists its contents are discarded and the file is treated as a new empty file. |
| "a+" | **append/update:** Open a file for update (both for input and output) with all output operations writing data at the end of the file. Repositioning operations (fseek, fsetpos, rewind) affects the next input operations, but output operations move the position back to the end of file. The file is created if it does not exist. |

# File Handling: fread & fwrite

```
size_t fread(void *ptr, size_t size,
             size_t count, FILE *stream);
```

- reads (size*count) bytes from stream; stores at ptr
  - does not check file sizes to check that enough memory is allocated
  - returns the amount of data actually read (to detect errors)

```
size_t fwrite(const void *ptr, size_t size,
              size_t count, FILE *stream);
```

- writes size*count bytes to stream from address given by ptr
  - also returns the amount of data actually written (to detect errors)

## (Buffered) fread Example

```
#include<stdio.h>

#define BUFFER_SIZE 100
int main(void)
{
    FILE* file_reader = NULL;
    char buff[BUFFER_SIZE];
    file_reader = fopen("test.txt","rb");

    if(NULL == file_reader) { printf("\n fopen() Error!!!\n"); return 1; }

     int bytes_read = 0;
    while ((bytes_read  = fread(buff,1,BUFFER_SIZE ,file_reader)) != 0)
    {    printf("Read %d bytes into buffer \n",bytes_read);
         printf("it starts with %c\n",buff[0]);
    }

    fclose(file_reader);
    return 0;
}
```

fread_buffered.c

# File Handling

Move the file read/write position:

```
int fseek ( FILE * stream, long int offset, int origin );
```

- Returns a 0 if successful, non-zero otherwise
- origin is given by

| | |
|---|---|
| SEEK_SET | Beginning of file |
| SEEK_CUR | Current position of the file pointer |
| SEEK_END | End of file * |

Find the current file position:

```
long int ftell ( FILE * stream );
```

Question: What does the following code produce?

```
FILE *file_handle = fopen("q8.c","r");
fseek(file_handle, 0, SEEK_END);
int variable = ftell(file_handle);
fseek(file_handle, 0, SEEK_SET);
printf("%d \n",variable);
```

File:q8.c

# Other file functions

- Functions that work with Standard Input / Output can work with files instead
  - usually prefixed with an 'f'
  - E.g., fscanf, fprintf
  - http://en.wikibooks.org/wiki/C_Programming/File_IO

```
int      fclose(FILE *);
int      feof(FILE *);
int      ferror(FILE *);
int      fflush(FILE *);
int      fgetc(FILE *);
int      fgetpos(FILE *, fpos_t *);
char    *fgets(char *, int, FILE *);
int      fileno(FILE *);
void     flockfile(FILE *);
FILE    *fopen(const char *, const char *);
int      fprintf(FILE *, const char *, ...);
int      fputc(int, FILE *);
int      fputs(const char *, FILE *);
size_t   fread(void *, size_t, size_t, FILE
*);
FILE    *freopen(const char *, const char *,
FILE *);
```

```
int      fscanf(FILE *, const char *, ...);
int      fseek(FILE *, long int, int);
int      fseeko(FILE *, off_t, int);
int      fsetpos(FILE *, const fpos_t *);
long int ftell(FILE *);
int      ftrylockfile(FILE *);
void     funlockfile(FILE *);
size_t   fwrite(const void *, size_t, size_t,
FILE *);
int      getc(FILE *);
int      getchar(void);
char    *gets(char *);
int      getw(FILE *);
int      remove(const char *);
int      rename(const char *, const char *);
void     rewind(FILE *);
```

## Binary files

- Files can be treated as text (default) or binary
  - No difference under *nix
  - but there is under windows

- e.g.; FILE *main_file = fopen(“data.txt”,”rb”);
  - opens in “binary” mode
  - ("rb", "wb", "ab", "r+b", "w+b", "a+b")
  - man fopen

- Text mode on windows…
  - https://en.wikipedia.org/wiki/Text_file
  - will convert **carriage return/line feed** combinations when read/written.
  - fseek may not work as expected
    - file may not be the same size as the data in it

- A file should be written to and read from in the same mode
  - or it may not work on all operating systems

## Debugging: Some reminders

## Debugging

Approaches to debugging:

- Stare at the code until the mistake is obvious
  - doesn't always work!

- Keep changing code until it works perfectly
  - surprisingly common, but not very effective

- **Active debugging**: take a systematic approach to finding out where the program does not do what is expected
  - You do need to find a 'test case' that shows the fault
  - Store your 'test  input' in a file and use the command line redirection
  - e.g. ./a.out < test1.txt

How OJ gets its input...!

(so use this to test your code if you get OJ weirdness!)

## Testing Strategies

- Make a number of test cases
  - Automate the testing process, e.g., with a test script
  - Test individual functions as well as the whole program
- Test all of the limits
  - What maximum and minimum values should the input take?
  - Include 0 values
  - Include negative values, if applicable
  - Create random test data – you may not be able to think of the right case
- Note and test all your assumptions
  - Every variable has a finite range – test the limits of this range
  - Particularly important for arrays
    - Include testing for 'off by 1 errors'

# Print debugging

- Print into a format you can view and filter
  - E.g. printf("loop,%d, i,%d, array[i],%d, array,%d\n",loop,I,array[i],array)
  - Redirect the output to .csv file, and open it with a spreadsheet
- Narrow down the problem
  - E.g., find out which half of the program gives an unexpected result
  - Then keep splitting in half again until you find the problem
- You can print to standard output (stdout) and error output (stderr)
  - printf goes to stdout
  - Can use `fprintf(stderr,"format string%d\n",parameter);`
  - Both are displayed on the screen, but can be redirected to different files
  - E.g.; `./a.out 1>output.txt 2>error.csv`

## Using a debugger

- Another solution is to use a 'debugger'
  - you already saw gdb (the gcc debugger)
  - allows you to step through the code and examine variables as you go
  - most development environments have sophisticated debuggers

- Programs need to be compiled with debug information
  - add "-g -o0" when compiling
  - relates the program to the source code that generated it

- See `gdb` example in remainder.
- Also: do not forget about `valgrind`!

## Common Mistakes

## Mistakes in Assignments

- Handing in:
  - no name on report or on code
  - no pdf file (but .docx or whatever...)
  - submitting not as .zip, but as .rar, .7z, ... etc.
    - use the **standard (pkzip) zip file format!**
    - https://en.wikipedia.org/wiki/Zip_%28file_format%29
    - supported by
      - winzip, winrar, etc. on windows/mac os X
      - 'zip' on linux (man zip)
  - test your zip file before submitting!

→ if we cannot open it, we cannot grade it ←

- Report:
  - do: "I used insertion sort, and ignored every odd number by checking modulo 2"
  - don't: "I watched this youtube video, and searched google"

## Mistakes in Assignments

- C errors:
  - indentation wrong, or mix with spaces and tabs
  - inconsistent naming convention
  - failure to use constants (e.g., const or #define for MAX_INPUT_SIZE)
  - reading input incorrectly…
    - not stopping at the right point `EOF`, or end of string (`\0`), or EOL (`\n`)
    - input that is received by your program ends with `EOF`
    - not knowing how to use `scanf`!
  - not freeing all memory
    - use `valgrind`!
- Design / debug problems:
  - not testing the program with more than the input given by the assignment
  - copying without understanding… NOT useful!

# More common mistakes

- Using memory that is not initialised
  - This includes arrays, allocated memory etc.
  - Can be difficult to find as there are no errors shown, and results are unpredictable.

- Reading / writing over the end of an array
  - both positive and negative indices
  - (or not allocating enough space)

- Not returning a value
  - This is not a compile error in C
  - (but use `-Wall` !)

# More common mistakes

- Empty loops or conditionals
  - `while (true); {…}`
  - `if (x < 0); {…}`
  - Can be difficult to see, even when you know where the problem is
- Mixing up == and =
  - `if (a = b) { … }`
- Mixing up variable types
  - Particularly pointers, but also other primitive types
- Passing parameters in the wrong order
  - Functions that take variable arguments (like printf) are particularly risky as there will be NO error checking

## Avoiding common mistakes

- Compile with all warnings on
  - gcc -Wall
  - **fix all warnings** you have!
- Immediately initialise all memory
  - small cost in performance
  - (can always disable later on…)
- Don't hard-code limits (e.g, sizes of arrays)
  - At the very least #define them to make the meaning clear
  - Insert error messages when limits are broken
- Annotate code with all limitations imposed / assumptions made
  - make clear to another (and yourself!) why some code may fail to function
- Break code into small functions
  - ideally test each function separately

## Avoiding common mistakes

- Consider breaking complicated expressions into multiple stages
  - Should make debugging easier
  - Limited/no impact on performance: compilers are very good in optimizing this!
- Test as you write
  - Don't wait until your program is complete before testing!
  - You may need to write a 'test harness' if your program is not complete
- Work out what exactly your code needs to do
  - Don't do more work than you have to
  - Consider reusing known algorithms or data structures

## Common mistakes and useful hints

- Arrays and Pointers are not, quite, the same
  - equivalence between *array indexing* and *pointer arithmetic*
  - when you declare an array, you also define the space for the array
  - declaring a pointer just declares space for the pointer

- That is:

```
char* somestring;  // (takes 4 or maybe 8 bytes)
char somestring[100]; // takes 100 bytes
```

- Arrays are indexed from 0
  - think about pointer arithmetic: how much you add on to the address
  - if you allocate `n` elements, the last element is at index `n-1`

## Strings

- A string is just an array of chars
  - WITH a terminating character (`'\0'`) to mark the end!
  - so: the last valid character in a string is at index `n-2`
- There is support for C strings within the language and the libraries
  - can do `char somestring[]="This is a string";`
- Make sure you declare enough space for the array and/or string!
- A string is an array of chars; so an array of strings is…

  1) An array of pointers to chars, or

  2) A 2-dimensional array of chars

  - syntax to access these is the same:

```
printf("%s \n",arrayofstrings[i]);
```

## Questions

## Summary

- You (should) know quite a bit about C!
  - some of the history: why C was invented, when you should use it
  - declarations, definitions, header files (and pre-processor)
  - input, output, files
  - memory organization:
    - instructions, static data, stack, heap
    - local vs. global variables
    - when to use what type of memory
  - arrays, pointers and their 'equivalence'
  - multidimensional arrays, and arrays of pointers
  - custom data types: structs, unions, etc.
  - dynamic data structures
  - testing, debugging
  - and... of course: a wide range of language elements
    - arithmetic operators, bitwise operators, logical operators, loops, conditions, switches, functions, ...

## Appendix: Further Topics & Resources

# Using GDB: Example

- Example:

```
File:d1.c
#include <stdio.h>
main ( )
{
float x=0,y=0;
y==1;
x=y +3/2;
printf ("%f\n" , x ) ;
}
```

- We expected 2.5, but the output is 1
- In this case, staring (and experience) may work – but we will demonstrate with a debugger!

  ```
  gcc -g -O0 d1.c
  gdb ./a.out
  ```

# Using GDB: Example

- What do we test first?
  - It depends! (Identify the likeliest cause of errors)
- Let's start at the end – does the correct value get passed to printf?

```
break 7  (the line number to stop on)
run  (start the program running, it will stop on line 7)
print x
$1 = 1
```

- So, x=1 at line 7 – there was a problem before that
- Try again

```
cont     (continue running – this time the program will finish)
break 6
run
print y
$2 = 0
```

# Using GDB: Example

`quit` exit the debugger

Fix the problem, and try again

```
File:d2.c
#include <stdio.h>
main ( )
{
float x=0,y=0;
y=1;
x=y +3/2;
printf ("%f\n" , x ) ;
}
```

- Now the output is 2.0, but that is still wrong

## Using GDB: Example

- A second approach – **watch** the variable
- You will find out when it changes

gdb a.out
break 1
run (actually until line 4, the first line of code)
watch x
c (short for 'cont')

  - *Breakpoint 1, main () at d2.c:4*
  - Old value = 3.73518348e-39
  - New value = 0

- That's ok

c

  - Old value = 0
  - New value = 2

- `x=y +3/2;` - which should be 2.5;
- we know y=1, hence 3/2 must also be adding 1

## Using GDB: Example

Remember that 3/2 will be treated as an Integer calculation!
We can modify it to work as floating point
(3.0 is NOT an integer value, despite being numerically identical to 3, which is!)

```
File:d3.c
#include <stdio.h>
main ( )
{
float x=0,y=0;
y=1;
x=y +3.0/2.0;
printf ("%f\n" , x ) ;
}
```

Try again, now the program works!

# GDB commands

| | |
|---|---|
| break | sets a breakpoint (a point in the program where the debugging will stop) |
| | e.g.; break 10 |
| | break main |
| clear | clears a break point |
| print | display the value of a variable |
| x | examine the memory at a given address |
| watch | The debugger will sstop when the given variable is changed |
| run | Start the program running |
| run < | Start the program running and take input from the given file |
| cont | continue the program running |
| step | step one line at a time. (If it's a function call, step into the function) |
| next | step one line – but step over the function call |

User Manual - http://sourceware.org/gdb/current/onlinedocs/gdb/

Cheat Sheet - http://darkdust.net/files/GDB%20Cheat%20Sheet.pdf

# Reference: Standard Libraries

- ctype.h
  - decimal digit or not: isdigit(c)
  - convert to upper case: toupper(c)
- string.h
  - length (does not count '\0'): strlen(s)
  - copy s2 to s1: strcpy(s1,s2)
  - compare: strcmp(s1,s2)
- stdlib.h
  - pseudo-random integer from 0 to RAND MAX: rand()
  - terminate program: exit(status)
  - system execution: system("ls")
  - convert string to integer: atoi(s)
  - storage allocation and deallocation: malloc(size)

    free(pointer)
- stdio.h
  - print formatted data: printf
  - read formatted data: scanf

https://en.wikipedia.org/wiki/C_standard_library

# Data Type sizes

**These sizes can be found in limits.h and float.h**

| | |
|---|---|
| CHAR_BIT | = number of bits in a char |
| SCHAR_MIN | = minimum value for a signed char |
| SCHAR_MAX | = maximum value for a signed char |
| UCHAR_MAX | = maximum value for an unsigned char |
| CHAR_MIN | = minimum value for a char |
| CHAR_MAX | = maximum value for a char |
| MB_LEN_MAX | = maximum multibyte length of a character accross locales |
| SHRT_MIN | = minimum value for a short |
| SHRT_MAX | = maximum value for a short |
| USHRT_MAX | = maximum value for an unsigned short |
| INT_MIN | = minimum value for an int |
| INT_MAX | = maximum value for an int |
| UINT_MAX | = maximum value for an unsigned int |
| LONG_MIN | = minimum value for a long |
| LONG_MAX | = maximum value for a long |
| ULONG_MAX | = maximum value for an unsigned long |
| LLONG_MIN | = minimum value for a long long |
| LLONG_MAX | = maximum value for a long long |
| ULLONG_MAX | = maximum value for an unsigned long long |
| FLT_MIN | = min value of a float |
| FLT_MAX | = max value of a float |
| DBL_MIN | = min value of a double |
| DBL_MAX | = max value of a double |
| LDBL_MIN | = min value of a long double |
| LDBL_MAX | = max value of a long double |

## Data Type sizes

- Example

```
int find_min(int* array, int length)
{
    int i = length;
    int min = INT_MAX;
    while (i-- > 0)
    {
        if (*array < min)
            min = *array;
        array++;
    }
    return min;
}
```