IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# C Programing Language

**MSC-INF101B**

**Panagiotis PAPADAKIS**

# Course 4

**Objectives:**
- Basics of code debugging
- Principal C programming tools
- Some more advanced notes

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Debugging in C

- It refers to the **inspection** of **code** during its execution in order to detect and correct **errors**.
- Preliminary diagnostics:
  - Using `void assert(int expr)`: evaluates `expr` and exits program if it is false by showing a message to `stderr` in the form of:

    *Assertion failed: expression, file filename, line number*

  - Using macros: `__FILE__`, `__LINE__` with fprintf(stderr, … ), e.g.:

```
fprintf(stderr, "Problem at %s:%i\n", __FILE__,
__LINE__);
```

If gcc is given the *-DNDEBUG* option or if we `#define NDEBUG` before the `#include <assert.h>`, then all asserts are disabled.

# Debugging – principal notions

- **Breakpoints**: specified places within the code where program **execution halts**. Places can be a specific **line number**, **function** call, **variable**, …
- **Display variable** contents
- **Modes** of code execution:
  - Line by line (*step over*)
  - Command by command (*step into*)
  - Until function return (*step out*)
  - Until next breakpoint or program end (*continue*)
  - ...
- Debugger mostly used **GDB** (GNU debugger)

https://www.gnu.org/software/gdb/

→ Compile and build your program with -g option to allow debugging

# Debugging – principal functionality

- GNU debugger use from **command line**:
  - Launching: `gdb -tui -q --args a.out arg1 arg2 …`
  - Breakpoints:
    - Adding breakpoints: `breakpoint function_name, b line_number`
    - List breakpoints: `info breakpoints`
    - Deleting: `delete bp_num, clear`
  - Watchpoints: `watch var1, info watchpoints`
  - Program running: `run, continue`
  - Variable display: `display var2`
  - Other common commands:
    - `next, step, return, finish, until`
    - Get help for a command: `help` *`command`*

# Make utility

- A GNU utility for *automatizing* the compilation and linking procedures of multiple files

  https://www.gnu.org/software/make/
- Requires a file called *makefile* that is given as argument to the `make` utility.
  - `make -f makefile`
- **Structure** of *makefile*:
  - **Comments**: lines starting with #
  - **Rules**:

```
target … : prerequisites …
       recipe
       …
```

  - **Variable definitions:** `CFLAGS=-I/myincludedir/ -DNDEBUG`
  - **Directives:** `include other_makefile`

# Make utility - rules

```
target … : prerequisites …
	recipe
	…
```

tab

One or more files

Name of file or action

Action(s) performed by `make`

Set of rules

```
#This is an example makefile
all: main.o functions1.o functions2.o
	gcc main.o functions1.o functions2.o -o main.out

main.o: main.c functions1.h functions2.h
	gcc -c main.c

functions1.o: functions1.c functions1.h
	gcc -c functions1.c

functions2.o: functions2.c functions2.h
	gcc -c functions2.c

clean:
	rm main.o functions1.o functions2.o main.out
```

“makefile”

► `make makefile` ⟶ Executes 1st rule (all)

# Make utility - variables

```
target … : prerequisites …
	recipe
	…
```

tab

One or more files

Name of file or action

Action(s) performed by `make`

Using variables

```
OBJS = main.o functions1.o functions2.o
CFLAGS=-O3
all: $(OBJS)
	gcc $(OBJS) -o main.out

main.o: main.c functions1.h functions2.h
	gcc -c $(CFLAGS) main.c

functions1.o: functions1.c functions1.h
	gcc -c $(CFLAGS) functions1.c

functions2.o: functions2.c functions2.h
	gcc -c $(CFLAGS) functions2.c

clean:
	rm $(OBJS) main.out
```

“makefile”

► `make -j4` → Allows parallel evaluation of 4 rules

# Documenting C - doxygen

- **Produces structured documented** code automatically from source files to html, LaTex, etc.
- Requires a **configuration file** that specifies how documentation will be produced. To generate it:

► `doxygen -g config_file`

- Then set various *TAGS* within your `config_file`: `PROJECT_NAME, EXTRACT_ALL, INPUT, FILE_PATTERNS, RECURSIVE, SOURCE_BROWSER, INLINE_SOURCES, OUTPUT_DIRECTORY, ...`

Finally, produce your documentation by:

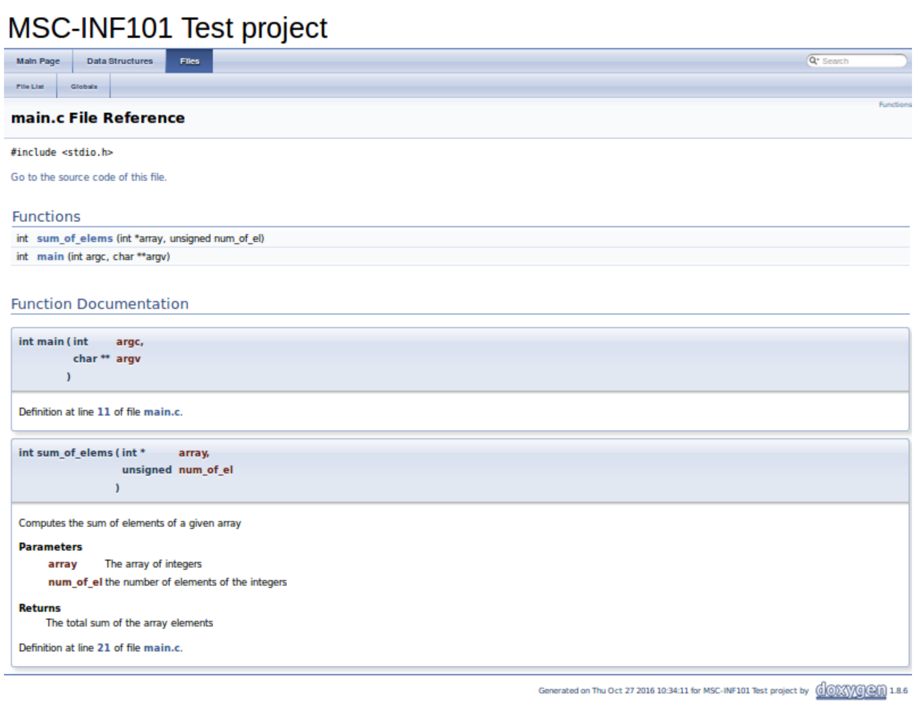
► `doxygen config_file`

# Documenting C - doxygen

- Sample file, with special documentation understood by **doxygen**

```c
#include <stdio.h>
/**
 * Computes the sum of elements of a given array
 * @param array The array of integers
 * @param num_of_el the number of elements of the integers
 * @return The total sum of the array elements
 */
int sum_of_elems(int *array, unsigned num_of_el);
int main(int argc, char **argv)
{
    int samples[] = {10, 5, 83, 23, 55, -2, 4, -9, 41, 70};
    int sumofelem;
    sumofelem = sum_of_elems(samples, 10);
    printf("The sum of the elements is %i \n", sumofelem);

    return 0;
}
int sum_of_elems(int *array, unsigned num_of_el)
{
    int i = 0, totalsum = 0;
    for(i=0; i<num_of_el; i++)
    {
        totalsum += array[i];
    }
    return totalsum;
}
```




Web-page (index.html)
produced by **doxygen**

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

[Try it yourself]

# Major C related libraries

- GNU C Library – glibc https://www.gnu.org/software/libc/libc.html
  - Extended range of C functions from various standards (e.g. ISO C, POSIX, BSD), allows development for system programming:
    - Files / Directories, Processes, String processing, Mathematical functions, Search – Sort, etc.
- GNU Scientific Library – GSL https://www.gnu.org/software/gsl/
  - Mathematical tools and algorithms
    - Algebra, Linear Algebra, Statistics, Transforms, Matrix operations, etc
- Other: DDD (debugger), GTK (graphical interfaces),...

# Advanced / extra issues

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# *Inline* functions

- Calling a function always causes a cost due to:
  - **Saving** the current **execution address**
  - **Jump to** the **address** where the **called function** is held and execute it
  - **Make the return jump** to the saved execution address

```
void swap(int *a, int *b)
{
    int t;
    t = *a; *a = *b; *b = t;
}
int main()
{
    /* some code */
    while(1)
    {
        /* some code */
        swap(&x,&y);
        /* some code */
    }
}
```

**The more** a given function is **called** within a program, **the more** this cost **impacts** performance

We can override this cost by declaring such functions as `inline`

```
inline void swap(int *a, int *b)
{
    int t;
    t = *a; *a = *b; *b = t;
}
```

The code of an inline function is inserted by the compiler into the body of the calling function

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Pointers usage

- **Pointers** can also used to refer to **functions**:
  - Example, quick sort function in `stdlib.h`:

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));
```

  - Useful when a small / atomic operation can take various forms
  - Useful when working together with colleagues

    (Use a pointer to a function whose body is not yet coded, but whose interface is known)

- Syntax of *reference* & and *dereference* operators * follows the same rules as for ordinary pointers.

# Pointers usage – qsort example

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

typedef struct student
{
    char st_name[256];
    unsigned short age;
}student;

int stud_compare(const student *stud1, const student *stud2)
{
    if((*stud1).age < (*stud2).age)
        return -1;
    else if((*stud1).age > (*stud2).age)
        return 1;
    else
        return 0;
}

void main()
{
    student *studarray = (student*)malloc(sizeof(student) * 3);

    studarray[0].age = 23; strcpy(studarray[0].st_name, "George Smith");
    studarray[1].age = 21; strcpy(studarray[1].st_name, "Sophia Bourne");
    studarray[2].age = 25; strcpy(studarray[2].st_name, "Peter Kern");

    qsort(studarray, 3, sizeof(student), stud_compare); /* Sorting of the array */

    for(int i=0; i<3; i++)
        printf("Student %i has name: %s and age %i\n", i, studarray[i].st_name, studarray[i].age);
    free(studarray);
}
```



Try it yourself

# Pointers' arithmetic

- Arithmetic operations on pointers are allowed (and very common)

| arr | | | | mystr | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 5 | | | | 6700 | | | 'H' | | Content |
| 1010 | ... | ... | ... | 4000 | ... | ... | 6700 | ... | Memory addresses (per byte) |

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

void main()
{
    int arr[] = {5, 13, 2, 99}, i;
    char *mystr;

    mystr = (char*)malloc(sizeof(char) * 6);
    strcpy(mystr, "Hello");

    for(i=0; i<5; i++)
        printf("Memory address: %x, Value %c \n", mystr++, *mystr);

    printf("\n");
    for(i=0; i<4; i++)
        printf("Memory address: %x, Value %i \n", arr+i, *(arr+i));
    free(mystr-5);
}
```

Program output

```
Memory address: 6700, Value H
Memory address: 6701, Value e
Memory address: 6702, Value l
Memory address: 6703, Value l
Memory address: 6704, Value o

Memory address: 1010, Value 5
Memory address: 1014, Value 13
Memory address: 1018, Value 2
Memory address: 101C, Value 99
```

Notice difference!

Try it yourself

# Dynamic memory allocation inside function

- How to pass uninitialized pointers as function arguments:

NOT working example

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

void my_mem_alloc(float *f_array, int n)
{
    int i;
    f_array = (float*)malloc(sizeof(float) * n);
    for(i=0; i<n; i++)
    {
        f_array[i] = (float)rand() / RAND_MAX;
    }
}

void main()
{
    int i;
    float *my_array;
    my_mem_alloc(my_array, 10);
    for(i=0; i<10; i++)
    {
        printf("Value of %i is %f\n", i, my_array[i]);
    }
}
```

Compiles but gives segmentation fault

Working example

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

void my_mem_alloc(float **f_array, int n)
{
    int i;
    *f_array = (float*)malloc(sizeof(float) * n);
    for(i=0; i<n; i++)
    {
        (*f_array)[i] = (float)rand() / RAND_MAX;
    }
}

void main()
{
    int i;
    float *my_array;
    my_mem_alloc(&my_array, 10);
    for(i=0; i<10; i++)
    {
        printf("Value of %i is %f\n", i, my_array[i]);
    }
}
```



[Try it yourself]

# Variable length arrays

- Since standard C99, we can declare variable length arrays, namely, arrays whose size is unknown when compiling

```
#include <stdio.h>

void main()
{
    int k;
    /* some code which operates on the value of k */

    float myarray[k]; /* Fine when compiling with C99 standard and above*/
    /* some code */
}
```

- The memory necessary for the array is allocated **statically**, which means that it is visible only in the local scope and cannot be modified

# Stream processing

- **Attention**: The moment of the *physical* I/O is decided by the OS and may NOT follow the order of program commands

```
#include <stdio.h>

void main()
{
    int k;
    printf("Hey there");
    k = 5;
    ...
}
```

May be printed on the screen LATER than the following commands

```
#include <stdio.h>

void main()
{
    int g;
    FILE *test_file;
    test_file = fopen("test.txt", "w");
    if(test_file != NULL)
    {
        fprintf(test_file, "Write me\n");
        g = 10;
    }

    ...
}
```

May be written on the file LATER than the following commands

Avoid overuse of `printf` for debugging for program (use `fflush` in that case)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

# Concluding points

- Change the order of logical expressions, according to expectation
- Precalculate repeated operations to speed-up code
- Favor the use of pointers; they help to conserve memory
- Use types appropriate to the semantics of your variables
- Make your code modular using functions and separate files
- Comment your code

- Further reading:

  **C in a Nutshell**, Ch.18-20