C Without a Spoon

Orna Agmon

Program of the Course

- About 8 units of course, with intervals of about a week.
- Within each unit, in the proper places, are interlaced Homework questions and exercises. Those will be discussed at the beginning of the next unit. Homework programs will be shown at the beginning of the next class, and discussed.
- References:
  - The C programming Language, second edition, Kernighan and Ritchie, has the final word, and is the place to look for answers.
  - man / info on a Linux system
- NOT everything will be explained in class - you will need to search for some more information in the book or man pages.

TOC

- Why C?
- C basics and Concepts
- CPP
- gcc compilation switches

What is C?

- A procedural language
- Compiled (not interpreted)
- No Garbage Collection - programmer maintains memory.
- CaSe SenSiTIvE != case sensitive
- Used as a basis for C++.
- Developed by Brian Keringham and Dennis Ritchie, when they needed a language to write UNIX with (was written in B)
- Can be linked with Fortran
- C gives you enough rope to hang yourself with.

Why C?

- Efficient - translates to machine language, no virtualization.
- Stable - Linux Kernel is based on it.
- Portable
- Has standards, many compilers including free ones.
- Strong checks, in many levels : Quality assurance.

C standards

- ISO C (**ANSI C**) since 1989. Widely supported, hence preferred standard.
- K&R (the first book on C by Kernighan and Ritchie) C. Old Standard. The book : The C programming language.
- C9X. Not widely supported by compilers.

Flow of a C Program

- A sentence ends with a semi-colon (;).
- New lines do not matter
- Still, it is more readable to break the lines at about 80 characters, though not compulsory.
- /* Comments must begin and end - Homework: think of a program that compiles, but produces the wrong executable, because of an unclosed comment*/
- //If the compiler is a C++ compiler, One liner comments are also valid.

Indentation and Syntax Highlighting

- Proper indentation helps reading structured programming.
- They are both done in good programming editors (emacs, xemacs (using tab), vi and others. Not pico.)
- Going over the program and indenting it helps finding blocks which were not properly closed.

Basic Variables

- Variables have some basic types: char, int, short, double, float.
- For logical variables use an integer-like variable.

Logical Operators

- Over variables:

  `< > == != || &&`

- Binary operations:

  `| &`

- Unary operation: `!`

- unrelated note: for a^b, use pow(a,b), and add the line `#include <math.h>` at the top.

Blocks

- An execution block is a single statement or a group of statements, enclosed by curly braces: {}. For example, an "if" statement, with and without blocks:

Blocks With and Without Curly Braces

- ```
  if (a<5)
      b=3;
  else
      b=2;
  ```
- ```
  if (a<5) {
      b=3;
      b=3;/*I can add more statements here*/
  } else {
      b=2;
  }
  ```

Where to put the new line?

- Even though C permits writing all the code in one line, it is good to go to a new line after an “if” or “else”, for debugging purposes.

Ternary If

- The same code can be written as:

```
b=(a<5)?3:5;
```

- Useful for initializations and clarity of code.
- Homework: write an expression that gives the minimal of the values a,b, using ternary if and regular if.

Short-circuit Evaluation

- `if ((a>0.0) && (sqrt(a)<5.0))`
  `b=3;`
- sqrt is not allowed non-positive input, but it will never be evaluated.

Do/While Loops

- ```
  int i=0;
  while (i<10) {
      ++i;
  }
  ```
- ```
  int i=0;
  do {
      ++i;
  } while (i<10);
  ```
- What is the difference?

The for loop

- ```
  int i,N=10;
  for (i=0;i<N;++i){
      /*execution block*/
  }
  ```
- 3 clauses in the parenthesis: initialization, a condition for keeping on the loop, and something to do when going to the next cycle.
- Within the block, comes the actual code to be executed.
- The loop variable must be declared before the loop, otherwise the behavior is not well defined (C++ extension).
  ```

Break and Continue

- `int i,j=0;for (i=0;i<10;++i){ if (i<5) continue; ++j;}`
- `int i,j=0;for (i=0;i<10;++i){ if (i<5) break; ++j;}`
- What is the value of j?
- What is the value of i?
- What is wrong with this question?
- Note the obfuscated c in the examples. Bad formatting!

Switch

```
switch (i){
case 1:
    a=3;
    break;
case 2:
    a=3;
    break;
case 213124:
    a=-1;
    break;
default:
    a=-1000;
}
```

- The break prevents continuing to the next code line.

Falling Through a Switch

- ```
  switch (i){
  case 1:
      /*fall through!!*/
  case 2:
      a=3;
      break;
  case 213124:
      a=-1;
      break;
  default:
      a=-1000;
  }
  ```
- When collating options, add the comment /*fall through*/, to indicate intention.
  ```

When to use a switch?

- Used when there are many different options - a switch is more readable.
- Used when there is a key, and there are only certain legitimate options. Using enum (an integer with specific values only, see later), the code will not compile if the switch does not deal with all possible values.

go to label

- ```
  if (bad_example!=0)
      goto L1;
  bad_example=1;
  L1:   bad_example=0;
  ```
- Usually, it is wrong to use goto...
- Except, when catching errors, and cleaning up (see later in dynamic memory)
  ```

printf - Simple Output

- The function printf prints to STDOUT.
- It take an arbitrary number of parameters.
- It gets a format and the values to print. Special characters must be escaped. Homework: `man 3 printf`.
- `printf(‘‘one is %d, a third is %g’’,1,1.0/3.0);`
- Homework: What is the difference between 1/3 and 1.0/3.0? How much is 1/3.0?

Hello World

- ```
  #include <stdio.h>
  int main(void){
      printf("Hello World\n");
      return 0;
  }
  ```
- Copy the program to file hello_world.c (can be copied from /orna/lectures/c/examples)
- Compile with `gcc hello_world.c`. Run with `a.out`.
- Homework: Use the simple Hello World Program and the constructs discussed so far, to compute the square of the numbers 1 - 100. for the square of 20, 30, 40 print 0. Implement once using if, and once using switch. Bonus: print square roots. (man sqrt for the include, compile with -lm for math library).
  ```

CPP - The C Pre Processor

Orna Agmon

The Pre Processor

A utility (/usr/bin/cpp) which goes over the C source files, combining code snippets into coherent files. Its purpose is:

- Make sure that code pieces are written exactly the same way all over the program.
- Macros
- Have code in the source file without seeing it.
- Exclude source from the file, pending on switches.
- Prevent bad compilation and linkage
- Portability
- Create several executables from the same source

What does cpp actually do?

- To see what the preprocessor does: `gcc -E hello_world.c`, or `cpp hello_world.c`.
- Try with -P for readability.
- Homework: when is it better to use gcc -E?

The Basic cpp instructions

```
#include <stdio.h>
#include ‘‘action.h’’
#define PROGRAM_VERSION 10
#undef PROGRAM_VERSION
#ifdef  PROGRAM_VERSION
#ifndef  PROGRAM_VERSION
#if !defined(PROGRAM_VERSION)
#endif
```

The Include Path

- When given a file in an #include command, cpp searches for it at a given path - the include path.
- The include path includes system directories, such as /usr/include and its subdirectories.
- System and compiler include paths differ on different systems. (Use autoconf to set them)
- When using quotation marks, the include path begins with the current directory.
- Additional directories can be added using the -I switch.
- Pre Processor switches can be given to the compiler - it passes them to the Pre Processor.

```
gcc -I/home/orna/include/ hello_world.c
```

Choosing The Include Path

- HomeWork: How do you choose between include with “” and with <>?
- What can happen if you get it wrong?

Protecting Headers

- A header is a file, which contains the prototypes of functions. Typically, the file's name ends with .c, and the compatible header's name is the same, only with a .h suffix.
- A proper header must contain all the information required to define those prototypes.
- To avoid multiple definitions, use the following form (in this example, for a header called myheader.h)

```
#ifndef MYHEADER_H
#define MYHEADER_H
/* Here comes the header for function myheader itself*/
void myheader(void);
#endif
```

Protecting Headers Safety

Homework: What happens if you sometimes write in the format #ifndef MYHEADER_H, and some other times, in the format MYHEADER_DEFINED?

Definitions

The scope of the definition is the syntactic scope from this point on, in the same file.

- Example:

```
#define PAI PI_M
#define PAI 3.14
#define EPSILON 1.e-10
#define SIZE_OF_ARRAY 100
#define malloc mymalloc
#define MY_NEW_MACRO_APPROVED
#define DEBUG_MODE
```

- to make sure the same constants are used throughout the program
- to enable quick changes of sizes.

- to set a size of an array, which needed at compile time.
- to trick the compiler, by replacing a library function with another, local, function. Note that in such a case, the definition must not appear before the definition of the function.

Homework - Replacing a Function

- Use #define to make the program (written in previous Homework), which prints square roots, print squares instead.
- Use the following function (/orna/lectures/c/examples/square_function.c):

```
double square(const double a){
    return a*a;
}
```

Macros

- A bad example for a macro definition:

```
#define MIN(A,B) A<B?A:B
```

- A good example:

```
#define MIN(A,B) (((A)<(B))?(A):(B))
```

- Using a macro:

```
i=MIN(3,5);
```

- Macros may also be protected, to avoid re-definition:

```
#ifndef MIN
#define MIN(A,B) (((A)<(B))?(A):(B))
#endif
```

- Homework: Bring an example where using the MIN macro is dangerous, and has unwanted side-effects.

MACROS are problematic

- Complicated macros can be created using continuation lines:

```
#define MIN(A,B) \
        (((A)<(B))?(A):(B))
```

- But: You cannot debug inside a macro!
- It is hard to tell when compiler errors come from within a macro.
- Indentation of a macro is not automatically done by IDEs.

Stringification

- an argument preceded by a # is expressed literally.

```
#define WARN_IF(EXP) \
do { if (EXP) \
        fprintf (stderr, "Warning: " #EXP "\n"); } \
while (0)
```

- The do-while are there to make it possible to write WARN_IF(ARG);
- Homework: info CPP (more updated than the man page). This example and the next are from info cpp.

Concatenation

```
struct command
{
  char *name;
  void (*function) ();
};

struct command commands[] =
{
  { "quit", quit_command},
  { "help", help_command},
  ...
};
```

Instead:

```
#define COMMAND(NAME)  { #NAME, NAME ## _command }

struct command commands[] =
{
  COMMAND (quit),
  COMMAND (help),
  ...
};
```

The Compiler

Orna Agmon

Compilers

- C compilers are available on most UN*X machines.
- The free C compiler is gcc.
- gcc is widely available over various platforms.
- The compiler we use defines the meaning of the C we write, because compilers differ in the implementation.
- Some compilers do not comply with some standards, some enlarge the language with specific features.
- Strict C rules ensure that the program will compile on many compilers.

gcc Compilation Switches

Switches follow the compiler's name with a "-" in the command line. For example:

- Language Options: -ansi
- Warning Options: -Wstrict-prototypes -Wmissing-prototypes -Wpointer-arith -Wcast-align -Wimplicit -Wmain -Wswitch -Wformat-Wchar-subscripts -Wreturn-type -Wcast-qual -Wcast-align
- -pedantic - do not use gcc extensions
- -Wshadow
- -Wunused
- -Werror

More gcc Compilation Switches

- Question: what does -Wall do? Compare $(0 == a)$ to $(a == 0)$ to $((a = 0))$. compare_asign.c
- definitions: -Dmacro=def, -Dmacro : define a macro all over the program.
- Profiling: -pg
- Debugging: -g
- output: -o targetname
- linkage: -c
- Optimization: -O, -O0, -O1, -O2, -O3, -finline-functions, -ffloat-store

Homework - gcc switches

- Read the section in gcc man page about Warning options, and specifically check what the above mentioned options do.
- Write a program which prints your name, only if macro YOUR_INFO is defined. Compile it twice from the same source, and run.
- Change the program such that is prints A,B,C, if YOUR_INFO equals 1,2,3 respectively, compile and run in various methods.
- Between the last two exercises, which was intended to use #ifdef?

gcc Extensions

Many gcc extensions are nowadays acceptable in many other compilers. Read all about gcc extensions under info gcc, C extensions.

- inline functions
- c++ comments

Profiling with gprof

- Compile with optimization data using -pg. Use file the_third_rule.c.
- Run the executable (a.out). A file called gmon.out will be created.
- Run `gprof a.out` to get a report on the performance.

Variables

Orna Agmon

Variable Declarations

- Variables are always declared at the beginning of a block (in C++, they can be declared anywhere within the code)
- Variables inside functions have a lexical scope they are valid in.
- Global variables (use only when must!) have a syntactic scope they are valid in.
- Best to declare variables as close (syntactically) to where they are used.
- Best to initialize variable when declaring them. An initialization can be done with constant values only. gcc extention: runtime values as well.
- Best to declare variables in the smallest lexical scope.
- Variables may shadow each other - avoid this (-Wshadow)

- Unused variables and parametersmay indicate a bug (-Wunused: parameter,variable,value)

Built in Types

- Built-in variables: float, double, char, int, long, long long (not always supported).
- C types specify minimal sizes only: int is at least 16 bit, long is at least 32 bit.
- Short is least 8 bit, but will still be so on 64bit architecture (too many things rely on this).
- gcc extention - an integer with defined number of bits:
  `unsigned int foo:1` for logical.

enum - a self generated type

- enum is a way to give integer values a literal meaning
- ```
  enum key_function_e{
      KEY_FUNCTION_NONE=0,
      KEY_FUNCTION_CONST=1,
      KEY_FUNCTION_CONSTANT=1,
      KEY_FUNCTION_LINEAR,
      KEY_FUNCTION_CUBIC,
      KEY_FUNCTION_TRIANGULAR
  }key_function;
  ```
- Enum values are set from 0 on, unless forced.
- Enum values are global - best to use a common initial to all enum values of the same variable.
  ```

Enum - cont.

- Enum values may be added in time - best not to rely upon their numeral value. If must, make a comment on the definition, and explicitly set the values.
- An enum value can be printed as an integer. To print it as a string you must write a function per each enum.
- z e2s.pl will create for you a file with functions which work both ways: converting enums to strings and strings to enums.

typedef

```
typedef enum key_function_e{
    KEY_FUNCTION_NONE=0,
    KEY_FUNCTION_CONST=1,
    KEY_FUNCTION_CONSTANT=1,
    KEY_FUNCTION_LINEAR,
    KEY_FUNCTION_CUBIC,
    KEY_FUNCTION_TRIANGULAR
}key_function_t;

key_function_t key_function1, key_function2;
```

typedef - cont.

- It is useful to define a new type when:
  - The type definition is complicated
  - A variable typed thus is passed into a function, and you want to change its const attribution.
  - Generalization and architecture independence: Usage of special types will insure consistency with system libraries: **size_t** for sizes, **time_t** for the return value of the **time** function.

Arrays

- An array of integers:`int a[100];`
- Access to an array is done using square brackets: a[0] is the first element.
- A multidimensional double array: `double b[10][34][20];`
- A multidimensional array is arranged in the memory such that b[i][j+1] follows exactly after b[i][j] (opposed to Fortran arrays).

Strings

- A string is an array of char variables: `char mystr[80];`
- A C string is NULL terminated: after the last significant character comes the character `'\0'`, whose value is really 0000000 in binary.
- C functions which work on strings are found in string.h.
- Homework: man snprintf, compare to sprintf. man strncat, strncmp, strncpy, strlen. When is strcpy usefull?
- Note the confusing return value of strncmp.

Pointers

- Each named variable is a nickname for an address in the memory.
- `&a` is the address of the variable a. The size of the number depends on the architecture - on a 64bit arch it will be a 64bit number.
- `int *b=&a` :a variable defined as a pointer to a type can hold its address.
- Refer to the value in the address held in a pointer using *: `*b` will give the value stored in variable a, even if a is set to a new value.
- Pointers have an alignment: a pointer for a small data type may be an invalid pointer for a larger data type.

Casting

- Casting is converting variables between types.
- An implicit casting is done for example when using sqrt(i), where i is int. Also when computing a mixed formula with integers and doubles, or when setting its output into an integer.
- `int i=1.0/3.0;`
- `double i=1/3.0;`
- Usually casting is dangerous.
- Because of pointer alignment, pointer casting is extremely dangerous, and may be unportable. For example, the space of one 8-byte float will accommodate 4 2-byte int values on some system, and 2 4-byte int values on others.
- Check using the program casting.c.

struct

- A struct holds together a collection of variables.
- ```
  struct linear_function_s{
      double a,/* slope of line */
  double b;/* constant of line */
  }linear_function;
  ```
- The order of the variables within the struct in the memory is according to their order in the definition.
- However, there may be gaps (padding) between the variables - according to implementation.
- Padding may be controlled by a compiler option.
- A struct does not have to be named, but it helps debugging and makes compilation checks possible.
  ```

Copying a struct

- For most variables, copying is done using `a=b`.
- Structs may have gaps inside them. So using = may work, but is not standard.
- If you intend to copy a struct, initialize its memory before you fill values in it:

```
memset(linear_function,0,
       sizeof(struct linear_function_s));
```

Then copy all the memory using memset or bzero:

```
memcpy(linear_function1,linear_function2,
       sizeof(struct linear_function_s));
```

- Homework: man memcpy, memcmp,memset, bzero.

Parts of structs and pointers

- ```
  struct linear_function_s linear_function, *ptr;
  double c;
  c=linear_function.a;
  ptr=&linear_function;
  c=ptr->a;
  ```
- An element of a struct is referred to using a period (“.”)
- An element of a struct to which you have a pointer is referred to using an arrow (“`->`”);
- The names of the parts of the struct are defined by its definition.
- A struct can be named a name you like.
  ```

A linked list

A linked list is a data structure intended for dynamic growth, as well as a dynamic change of order of the elements. Though there can be many implementations, most of them use a struct:

```
typedef struct list_element_s{
    double payload;
    struct list_element_s *next;/*a must*/
    struct list_element_s *prev;/* may be needed*/
} list_element_t;
```

Homework: Why did I not use the typedef here, but wrote three times the long name of the struct?

Functions

Orna Agmon

Defining Functions

- There are no procedures - only functions.
- A function can have side effects, apart from the return value. (procedural programming)
- A function is defined by its input (arguments) and by its output (return value):

```
int square(int a, int to_print){
    int b=a*a;
    if (to_print)
    printf(``%d\n'',b);
    return b;
}
```

Return Value

- A function can have side effects.
- A function can return a void, which means it does not return a value.
- The return value of a subroutine can be used as a flag for failure. "Success has many parents, but failure is an orphan". Still, success has little info to pass, and failure can be caused due to many reasons. Hence, 0 is used for success, non-0 for failure. In cpp implemented using throw/catch.

Static Functions

- To define a function only in syntactic scope of the same file, use static:

  ```
  static int square{int a, int to_print};
  ```

  static functions cannot be profiled - use a macro instead.

System calls and Library functions MAY FAIL

- System calls are system functions - to communicate with the operating system
- Homework : write a program which removes a file in two ways: 1.using the unlink system-call, 2. using system. Man system, unlink.
- Library functions are functions supplied by the c language or other packages. For example, the math library supplies sin, cos.
- The return value of system calls must always be checked, for the program to be robust.
- Find about the return values at the end of the man/info page.
- Homework: read man page and find return values for examplatory functions for example: fopen, fclose.

errno

- System calls and some library functions set a variable called errno when they have a problem, to state the type of problem.
- Usually, when a system call fails, it sets errno to negative value.
- A succesful library call is allowed to change the value of errno. (After a successful call, errno is undefined).
- If -1 is a legitimate value of the syscall, it may be necessary to initialize errno with 0, to know if it failed.
- strerror translates the error code to a string. perror prints it.
- Homework : man errno, strerror, perror.

Prototypes

- A prototype resembles the definition of the function, only with a semi-colon at the end:

```
int square(int a);


int square(int a){
    return a*a;
}
```

- The prototype is usually placed at the header file.
- If the file which includes the function includes the header as well, the compiler can verify they match.
- Then any other code which uses the function can rely on the header for accurate usage.

Homework - Multiple files in a project

- Write a program with two files. In each file is a function. The main function calls a function in the other file.
- Compile using a standard portable makefile, using the following prodedure:
  - Have configure.in, makefile.in in the directory (you can copy from an existing program).
  - autoconf
  - bring needed scripts for configure (it will complain specifically if you do not)
  - configure
  - make

Const, Addresses and Values

Using const on a parameter or a part of it means that the code will not compile if the function tries to change the constant part.

- Pass by value:

```
int square{int input};
int square{const int input};
```

- Pass a pointer to value:

```
int square{int *input};
int square{int * const input};
```

- A variable may be declared const within a function, and be used instead of a #define-ed value. (It has a lexical scope just like a regular variable).

Passing a pointer

- Inside the function, *input is the value itself.
- input is the address of the value.
- `*input=5;` will actually set the variable outside the function.
- `input=NULL;` will set the address inside the function, but will not affect what happens outside.
- When using const and trying to change the value inside the function it will not compile. When not using const, it may cause a bug.
- In order to pass the adress of variable b to a function, pass &b.

Const - Homework

- Write a program in which you attempt to change a variable which is passed by value.
- Print its value at the end of the function.
- Print its value outside the function.
- Write a program in which you pass a const value and try to change it in the function, and compile it. What happens?

Dynamic Memory

Orna Agmon

Memory Areas

- Stack - ordered according to order of access into functions. If the language allows for recursive functions (like C), each function entrance is on another place on the stack. C stack is rather small compared to fortran, and platform dependent - place your big arrays some place else! (If you do not, you may overrun your stack pointer). Size of arrays must be known at compile time.
- DATA - Global variables
- Heap - Random access, according to available memory at the moment of request. Size of arrays may be only known at some point on run time. Place your large arrays here!

Dynamic Allocation

- malloc requests memory from the heap. free returns it.
- Homework: man free, alloc, realloc.
- Naive allocate and free:

```
int *a=(int *)malloc(4*sizeof(int));
a[3]=3;
free(a);
```

- Why Free at all? (compare and contrast to garbage collection)
- Allocation does not always succeed. Nor does Free. When can they fail? Why?

Clever Allocation

Code according to Nadav Har'El.

```
void * my_alloc( const size_t size){
    void *mem;
    if(size<=0){
        printf ("Invalid attempt to allocate zero bytes");
        abort();
    }
    if(!(mem=malloc(size))){
        printf ("Allocation failed");
        abort();
    }
    return mem;
}
```

Clever Free

Code according to Nadav Har'El.

```
void my_free(void ** const pointer){
    if(*pointer==NULL){
        printf("Invalid attempt to free a NULL pointer");
        abort();
    }
    free(*pointer);
    *pointer=NULL;
}
```

Usage of clever dynamic memory management

- Use a macro to replace all your memory allocation calls:

```
#define ckvalloc(ptr,type,nelem) \
((ptr)=(type*)ckalloc((nelem)*sizeof(*(ptr))))
```

- Homework: write a program which allocates and frees memory using the previous code.

Memory Violations

- Memory smear - reading from an unintended and unrelated location, or worse - writing there.
- Memory Leak
- Read access to uninitialized memory
- Should I free the program's memory upon exit?
- Automatic tools help avoid those problems:
  - valgrind on Linux on i386
  - third on OSF1 on alpha
  - zmalloc on all platforms - compiled with the code

Private Memory Management via pointer arithmetics

- ```
  const int n=10;
  int *x=(int *)my_malloc(2*n*sizeof(int));
  int *y=*x+n;
  ```
- What is good about it, what is bad? (relate to memory fragmentations, memory violation, efficiency)
- How should this memory be freed?
- What is y[0] if x was freed?
- What happens if we free y?
  ```

Memory fragmentation and the Intervals tree

- The info about the heap is saved (in Linux) in the format of an interval tree: a binary tree, where leaves on a node are kept as long as there is a difference in their status.
- The intervals tree may be able to give small bits of memory, but not a large piece, if the memory is fragmented.
- The intervals tree may be better at uniting free memory than your program.
- Manage the memory only when you have a good reason to believe you know better.

Good Programming and Advanced Programming

Orna Agmon

Efficiency

- Profiling will clear the big messes, but not the small ones.
- Mis-concept: computation is slower than memory access
- Current fact: memory access is the bottle-neck of high performance computing.
- Hence, do not try storing small parts of calculations only because you use them twice.
- However, do store them in meaningfully named variables, if it helps readability.
- The compiler can optimize such things.

Cache misses

- The cache is a fast small memory, located close to the CPU or in it.
- Memory access is "cheap" when the data is already in the cache.
- When a line is brought to the cache, more data than the variable you asked for is brought to the cache. (near data)
- When designing a program, it is worth considering how to have minimal cache misses: work for a long time on the same piece of data, instead of going over arrays, and throwing everything from the cache right after.
- valgrind can profile cache misses.

Documentation - What Goes Where?

- Document exactly once: double documentation calls for contradictions.
- The documentation of the general behavior of the function comes in the header file, near its prototype. Describe its input and output and its general usage. Intended for users of the function.
- Before the implementation of the function, describe its general algorithm. Intended for people who read the code (possibly briefly).
- When documenting the code itself, avoid obvious statements (increasing i by 1).

Documentation - How?

- 1:1 is considered a fair documentation: code ratio
- Do not fear to leave empty lines for readability.
- Doxygen generates an html help from comments in the code. This prevent the need to document twice.

Debugging

- Print to a file, to the screen (STDOUT) or to the standard error (STDERR) - printf, fprintf
- WARN_IF macro
- abort() - to dump a core, and later debug using a picture of the memory as it was the the violation was commited (use gdb -c core to see it)
- IEEE functions: isnan, isinf, isalpha etc. They consider locale.
- Every command begins on a new line, to enable breakpoints.
- Code which is excluded using macros should be compiled on its own, is possible, to be able to check every combination of macro switches.

Debuggers

- -g for debug symbols
- valgrind –gdb_attach=yes to attach to the process when a emmory violation was commited.
- Debuggers:
  - gdb (and graphic ends: ddd, xgdb)
  - ladebug (and a graphic end: xladebug)
  - dbx

Linking Fortran and C - matching variables

- Fortran actually accepts pointers to variables - this is how in Fortran, variables can be changed inside the function.
- When Fortran needs to get $integer * 4$, or an array of it, C can pass `int *`, on architectures where int is 32 bit.
- When Fortran needs to get $integer * 2$, or an array of it, C can pass `short *`, on architectures where short is 16 bit.
- When Fortran gets $real * 8$, or an array of it, C passes `double *`.
- When Fortran gets $real * 4$, or an array of it, C passes `float *`.
- When Fortran needs to get $integer * 4$, C needs to pass `int *`.
- Multidimensional arrays are held transposed. If Fortran needs A(3,4), C needs to passes A[4][3] (multidimensional array - not pointer to pointer!).

Linkage

- After compilation, every function/subroutine gets a symbol.
- The symbol is not related to the name of the file in which the subroutine was written.
- The name is related in various ways to the subroutine's name.
- When linking several languages, name-mangling may affect the programming: The calling language may have to use a different name for the called function than the name which was defined in the code. When using only one language, it has no effect.

Name Mangling

- The other language's symbols usually have an underscore at their end. For example, subroutine mysub in Fortran may be referred to as mysub_.
- Depending on the Fortran compiler, there may be a different treatment for subroutine names wich include underscore inside them: they may get two underscores at their end. For example, subroutine my_sub in Fortran may be referred to as my_sub__.
- Since Fortran is not case sensitive, it cannot tell the difference between c functions which differ by caps. For example, if you have two c functions, `void myfunc(void)` and `void MYFUNC(void)`, and in the Fortran code MyFunc_ was called, which of the functions should be used?

Name Mangling - Answer

- The linker is activated by the compiler which compiled all the objects to one executable. If the compiler is Fortran, it will refuse to link two functions by the same name.
- Homework: what if the compiler which calles the linker is C? (To check this, create a main c function, which calls a fortran subroutine, which calls MyFunc_. Also compile two c functions, myfunc and MYFUNC.)

Linking Fortran and C: Conclusions

When a function is about to be called from another language:

- It must not include underscores.
- It had better be all lowercase and have no confusables.
- If it is Fortran, it cannot be get or return a logical (which C does not support).
- If it is Fortran77 it cannot get/return a pointer to pointer (which fortran 77 does not support).
- Still, there may be platforms where this is not enough and the code must change in order to link in this particular environment. Define the functions names as pre-processor constants at one place, to be able to replace them. For example:

```
#define FORTRAN_MY_FUNC myfunc_
```

Design

- Define constants using names, at one point
- Prepare to add more variables to the module -it easy ans safe. (for example: prefer allocating an array of structs over allocating arrays within a struct)
- Prefare interfaces that do not change when you add more variables to the module: for example, use a struct which holds the data to the function.
- string sizes should be fixed in one place
- Verify interfaces at compile time, or else at link time or compile time.
- Collect a (new) module's functions in one place. Within the existing code, add only calls to the new functions.

Input

- Separate code and data: Input should be read from files or from the command line - do not insert it in the code itself.
- When reading an input file, it should be self documented: for example, read first the number of lines, and then you know how many lines to read. Do not expect to get this data elsewhere. Better to have a mnemonic description of the data inside the data file.
- Save ASCII files, not binaries.

Command Line Input - Declaration

```
#include <getopt.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    char c;
    int int_after_t;
```

Command Line Input - Parsing

```
while((c = getopt(argc, argv, "nt:")) != -1){
    switch(c){
    case 'n':
        printf(``n was read\n'');
break;
    case 't':
        int_after_t=atoi(optarg);
break;
    default:
        printf("Usage:\%s [-n] [-t an_int]'',argv[0]);
        return -1;
    }
}
```

Functions with a varying number of arguments

- Useful for functions which print, like printf.
- The function declaration contains ... after its list of regular arguments, for example:

```
#include <stdarg.h>
void my_print(const char *format, ...){
    va_list args;
    va_start(args,format);/*Initialize the va_list,
            called with the last known parameter.*/
```

- The macro va_arg(args, type) pops the next variable in the list.
- va_end(args) handles the return from a function which called va_start. Homework: man va_start. Write a function which prints ints and doubles using a format.

Function Pointers

From man qsort:

```
#include <stdlib.h>

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

Qsort expects a function pointer, to a function which receives two void pointers and returns an integer.

Using qsort

```
int compare_ints(const void *a,const void *b);
int compare_ints(const void *a,const void *b){
    const int A=*(const int *) a;
    const int B=*(const int *) b;

    if(A< B) return -1;
    if(A==B) return  0;
    else return 1;
}
void sort_int(int *s,int n){
    qsort(s,n,sizeof(int),compare_ints);
    return;
}
```

Homework - Function Pointers

```
void myfunc(void){
    printf(‘‘Hello World’’);
    return;
}

int main(void){
    myfunc;
    myfunc();
    return;
}
```

what will be printed? Why?

Re-entrant functions

- A function which does not save its status in global variables or static variables is state-independent.
- State-independent functions are re-entrant.
- State independent functions can be used for recursion.
- State-independent functions are safe to use with threads. State independent design is easier to make parallelize later on, even if you do not consider more CPUs right now.

Recursion (recursion.c)

```
#include <stdio.h>
static void recursive_print(const char * const str);
int main(void){
   recursive_print("Hello World\n");
   return 0;
}
static void recursive_print(const char * const str){
   if (str==NULL) return;
   if (str[0]=='\0') return;
   printf("\%c",str[0]);
   recursive_print(str+1);
   return;
}
```

Programming Concepts

- Procedural programming: the basic units are procedures: execution units. return values are of lesser importance.
- Functional programming: basic units are functions, with no side effects.
- Object Oriented programming (OOP): basic units are variables. Variables can have functions to work on them

Object - Oriented Concepts

- Objects and their methods
- Modularity
- Encapsulation - access to private data of the object is limited, and done by special methods.
- Inheritance - the properties of an object can originate from properties of a similar, less sofisticated object.