Some Topics on the C Language (part I)

---

1. Consider the program `hello.c`:

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

Check what happens when only the preprocessor part of the C compiler is executed:

```
$ gcc -E hello.c
```

To inspect the Intel x86 assembly code produced by the C compiler for the program you can execute:

```
$ gcc -S hello.c
```

Check the result in the file `hello.s`.
Execute the following commands and observe the results:

```
$ gcc hello.c
$ gcc -o hello hello.c
$ gcc -Wall -o hello hello.c
```

You should always compile a C program with the `-Wall` option so that the compiler always presents *warnings*. These are problems detected by the compiler that, despite not blocking the compiler from generating an executable file, may give rise to runtime errors. You should correct these as if they are full fledged compilation errors.

To use the C *debugger*, an auxiliar program that allows you to run the executable file generated by the compiler one step at a time and to inspect the state of the program variables. To include debugging support you must use the `-g` option.

```
$ gcc -g -o hello hello.c
$ gdb hello
gdb> break main
gdb> run
gdb> next
gdb> ...
```

2. Consider the program `trig.c` that calculates tables of values for the trigonometric functions $\underline{\sin x}$ and $\underline{\cos x}$ for integer angles given in degrees (from 0 to 360).

```
#include <stdio.h>

#define START     0
#define ONE_TURN  360

double cos_table[ONE_TURN];
double sin_table[ONE_TURN];

void build_tables() {
   int i;
   for (i = START; i < ONE_TURN; i++) {
      sin_table[i] = sin(M_PI * i / 180.0);
      cos_table[i] = cos(M_PI * i / 180.0);
   }
}

double sin_degrees(int angle) {
   return sin_table[angle % ONE_TURN];
}

double cos_degrees(int angle) {
   return cos_table[angle % ONE_TURN];
}

int main() {
   build_tables();
   printf("sin(20) = %f\n", sin_degrees(20));
   printf("cos(80) = %f\n", cos_degrees(425));
   printf("tan(60) = %f\n", sin_degrees(60) / cos_degrees(60));
   return 0;
}
```

Compile the program with the command: `gcc -Wall -o trig trig.c`. The compiler complains about some problem and does not generate an executable. Can you understand why? (hint: pay close attention to the error messages).

Correct the error and compile the program again with the same comand. The compiler complains again? What is the problem this time? How can you solve it? (hint: run the commands `man sin` or `man cos`).

3. Consider the following program, `pointers1.c`, that aims to exemplify some aspects of the use of pointers in C, specifically, the operators `&` ("address of") and `*` ("content of address").

```
int main() {
  int i, j, *p, *q;
  i = 5;
  p = &i;
 *p = 7;
  j = 3;
  p = &j;
  q = p;
  p = &i;
 *q = 2;
  return 0;
}
```

Compile the program with the command: `gcc -Wall -o pointers1 pointers1.c` and watch what happens to the variables by adding the following line at different points in the program:

```
printf("i=%d, j=%d, p=%p, q=%p\n", i, j, p, q);
```

Make a drawing that represents the system memory that shows the creation of the variables `i`, `j`, `p` and `q` and follow the execution of the program by changing their values in the drawing.

4. Consider the program `char_array.c` that moves through an array of characters:

```
#include <stdio.h>

int main() {
  int i;
  char msg[] = "Hello World";
  for (i = 0; i < sizeof(msg); i++) {
     printf("%c <--> %c\n", msg[i], *(msg + i));
  }
  return 0;
}
```

Compile it and execute it. How do you explain the result? Variable `msg` behaves as if it is of what type? Each increment of `i` corresponds to how many bytes?

Consider the program `int_array.c'`.

```
#include <stdio.h>

int main() {
  int i;
  int primes[] = {2, 3, 5, 7, 11};
  for (i = 0; i < sizeof(primes)/sizeof(int); i++) {
     printf("%d <--> %d\n", primes[i], *(primes + i));
  }
  return 0;
}
```

Compile it and execute it. How do you explain the result? Variable `primes` behaves as if it is of what type? Each increment of `i` corresponds to how many bytes?

5. Consider the programs, `call_by_value.c`:

```
void swap(int n1, int n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
}

int main() {
  int n1 = 1;
  int n2 = 2;
  swap(n1, n2);
  printf("n1: %d n2: %d\n", n1, n2);
  return 0;
}
```

and `call_by_reference.c`:

```
void swap(int *p1, int *p2) {
  int temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}

int main() {
  int n1 = 1;
  int n2 = 2;
  swap(&n1, &n2);
  printf("n1: %d n2: %d\n", n1, n2);
  return 0;
}
```

Make a drawing that represents the system memory that shows the creation of the variables n1, n2, p1 and p2 and follow the execution of the program by changing their values in the drawing.

Can you understand the difference between the two programs? Why is it that in the second program the values of n1 and n2 are swapped, unlike what happens in the first program?

6. Consider the programs bad_pointer.c:

```
#include <stdio.h>

int* get_int() {
  int i = 2;
  return &i;
}

int main() {
  int* p = get_int();
  printf("integer = %d\n", *p);
  return 0;
}
```

and good_pointer.c:

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

int* get_int() {
  int* p = (int*)malloc(sizeof(int));
  *p =2;
  return p;
}

int main() {
  int* p = get_int();
  printf("integer = %d\n", *p);
  return 0;
}
```

Compile and execute them. Note that, in the case of bad_pointer.c you have to compile with option -w to disconnect all *warnings* from the C compiler. Why? Can you understand what is happening?

Recompile the program `gcc -g -w -o bad_pointer bad_pointer.c` and run it with gdb.

```
$ gdb bad_pointer
gdb> break main
gdb> run
gdb> step
gdb> ENTER
gdb> ...
```

Where is the error? Why?

Note. Several scenarios can give rise to errors in the access to memory during the execution of a program. Such errors are usually reported by the operating system as `segmentation fault` or `bus error` and result invariably in the abrupt interruption of the program. Most commonly, these scenarios result from an improper use of pointers, namely from trying to apply the `*` operator to an invalid pointer. The following C snippets show three typical errors:

```
/*
 * Null Pointer: o endereço NULL não é válido
 */
char *p1 = NULL;
...
char  c1 = *p1;  /* erro em runtime */

/*
 * Wild Pointer: p2 não foi inicializado e tem um endereço inválido
 */
char *p2;
...
char  c2 = *p2;  /* erro em runtime */

/*
 * Dangling Pointer: um apontador que deixou de ser válido
 */
char *p3 = (char*)malloc(sizeof(char));
...
free(p3);
...
char  c3 = *p3;  /* erro em runtime */
```

7. Copy the following code to a file `trace.c` and compile it using `gcc -rdynamic -Wall trace.c -o trace`. Observe the output and try to explain it based on the structure of the program. Then, add this function:

```
void f4(void) { print_trace("f4"); return; }
```

and replace function `f3` by:

```
void f3(void) { f4(); print_trace("f3"); return; }
```

What happens then? Why?

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

#define MAXSYM 32

/*
 * Obtain a backtrace and print it to stdout. Adapted from:
 * https://www.gnu.org/software/libc/manual/html_node/Backtraces.html
 */
void print_trace(char *func) {
  printf("------------ BEGIN (%s) ---------------------\n", func);
  void *array[MAXSYM];
  char **strings;
  int size, i;
  size = backtrace(array, MAXSYM);
  strings = backtrace_symbols(array, size);
  if (strings != NULL) {
    printf("Obtained %d stack frames.\n", size);
    for (i = 0; i < size; i++)
      printf("%s\n", strings[i]);
  }
  free(strings);
  printf("------------ END (%s) ---------------------\n", func);
  return;
}

/* Threaded function calls to show the backtrace */

void f3(void)   { print_trace("f3"); return; }
void f2(void)   { f3(); print_trace("f2"); return; }
void f1(void)   { f2(); print_trace("f1"); return; }
int  main(void) { f1(); print_trace("main"); return 0; }
```

8. Consider the following C code fragments that make use of pointers. Explain how the pointers are being used, what part of the process address space is being used, what type of information is being pointed at and if the operations are safe (i.e., do not result in a `segmentation fault` or a `bus error`).

```
(1)
...
void f() {
  int x;
  g(&x);
}
...

(2)
...
int* f() {
  int x;
  return &x;
}
...

(3)
...
int* f() {
  int* x = (int*)malloc(sizeof(int));
  return x;
}
...

(4)
...
int g(int (*h)(int), int y) {
  return h(y + 2);
}

int f(int x) {
  return x*x;
}

int main() {
  printf("value: %d\n", g(f,2));
  return 0;
}
```