Institut für Informatik — TUM
Scientific **C**omputing in **C**omputer **S**cience
R. Wittmann, G. Buse

WT 2013

Tutorial
(Advanced Programming)
Worksheet 5:

Assignment 1: Variable properties

In this exercise we want to study how the compiler arranges variables in memory. Note that it might be especially interesting to do these tests in different environments (linux, windows, mac osx, *bsd) or even on different architectures (arm, ppc, sparc, x86, x86-64) and compare the results. Implement a piece of code following these four steps:

1. Define a new function which determines the byte alignment $(1, 2, 4, \ldots, 2^n)$ for a given (void*) address.
   Use C++ style casts (`reinterpret_cast` , `dynamic_cast`, `static_cast`) to convert the address to `size_t` on which you may perform arithmetic operations.

2. Define another arbitrary function (without return value) and declare several local variables inside. Make sure you have at least one **int**, one **double** and one **char**.

3. Now, for every variable in your function, print its address, its current value, its alignment and its size. Further, print the address of the function itself. **Hint:** The function address is not "1".

4. Depending on the number of variables you have chosen before, this might lead to a lot of copy and paste. As an exception, we will define a macro (**do not use macros!**) to save some implementation time. Replace all the output calls you made before by the following function macro:

```
#define PRINT_VARIABLE_DETAILS(var) std::cout \
  << "argument name is " << #var << std::endl;
```

Try different compiler optimizations, including `-Os`, `-O0`, up to `-O3` and architecture information using `-march=native` (for gcc compiler) or `-xHOST` (for icc compiler). Answer the following questions:

- Do you recognize a certain alignment pattern when looking at variables of the same type? What about different types?
- Do the function and variable addresses change when calling the program several times? Why (not)? Think about where variables "live" and how addresses are assigned.
- How does the order of the variables change and is this behaviour beneficial? Did you expect a different ordering? Discuss!

- How do the properties of the variables change with compiler optimization? For that, just go to the command line, compile your program first with `-Os`, run it, compile with `-O0`, run it, and so on. Again, what do you observe, and is this beneficial? Discuss!

Assignment 2: Functions and Memory Consumptions

After having analyzed how variables are mapped to memory, we will now try to estimate how much memory is occupied when we call a function. Again, the actual values heavily depend on the chosen environment.

1. Define a recursive function which
   - returns a `size_t` summing up the used memory on the recursive stack
   - takes the recursion depth as formal parameter for control over the stack depth
   - takes a reference address as formal parameter for deriving the memory footprint of the stack
2. To be able to adjust your memory consumption, declare *only* a local fixed-size byte array in your function:

   ```
   char a[PAYLOAD_SIZE];
   ```

   where PAYLOAD_SIZE will be specified to the preprocessor at compile time via a −DPAYLOAD_SIZE=<mysize> compiler flag.

Think about how function calls alter the callstack and how the variables of the calling function might be preserved between (recusive) function calls. Use your "tools" from Assignment 1 to analyze the resulting memory layout of the function calls. Finally, try different compiler flags and analyze the different outputs.

- How do the memory addresses of our variables change across function calls?
- Does the cost change if you change the payload size?
- Based on your findings, try to give a detailed explanation how the call stack works and how it is actually organized in memory.