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

WT 2013

Tutorial
(Advanced Programming)
Worksheet 11:

This week's exercises are all in the context of dynamic polymorphism and class inheritance. We will see how theses concepts are used to build complex models in software, and we will start to analyze how the compiler translates these models into lower level constructs.

Assignment 1: Modeling via Dynamic Polymorphism

This exercise shall reveal to us the benefit of proper modeling in object oriented programming via inheritance. We will use the free header library "CImg"[1] to render our objects and give us instant visual feedback.

The goal is to use the *dynamic polymorphism* in C++ to model a set of geometric shapes, in particular polygons, circles and ellipses. These, we want to draw afterwards and watch the results. Our abstract base class will be `Shaped2d`. It bears merely one property, its color, but it is further defined by its interface of purely virtual functions. These functions are for instance used to render objects of type `Shape2d` to our application's canvas. The canvas is a *singleton* of type `Canvas`, which can be created, obtained and deleted by several static members of the `Canvas` class.

In order to render objects of type `Shape2d`, we need to inherit from the base class and implement all virtual funcions in the inheriting classes. This is where other properties may be defined as well. As an example, see the `Polygon` class.

Your task in this exercise is to fill in the missing parts in the code skeleton "inheritance.cc" provided in Moodle. Look for the `TODO`s in the source code and follow the instructions. There is nothing to do in the `main` function itself, and once you complete the task you will see the results of your modeling.

Compilation

As usual, compiling the source code requires the switch `-std=c++0x` or `-std=c++11`. Furthermore, CImg depends on several system-specific headers and libraries. Under Ubuntu linux only the package `libx11-dev` is required, which is typically installed in the system path by default (even as a static library). Therefore, the only thing you have to do is pass the following flags to the linker: `-lm -lpthread -lX11`
For your convenience, we have included the header file "CImg.h" with the code skeleton, so just unpack it and you are ready to go. If your system satisfies the above conditions, the code compiles as is. MAC OS and Windows are supported

[1] `http://cimg.sourceforge.net`

by CImg, too, although other prerequisites will have to be fulfilled. Check the website for more information, it should be easy as well.

Assignment 2: Technical details

Similar to one of the previous exercises where we analyzed the call stack of a function, we will use the same methods to gain a deeper understanding of the C++ implementation. While the basic concepts are very similar on each platform, the actual addresses, offsets or sizes are highly dependent on your target system and platform. Use the file "find_vtable.cc" provided in Moodle as a starting point for your analysis.

- Your first task in this reverse engineering process is to look at the object sizes of our predefined class types. What is your expectation and do you notice anything special?
- As the classes require some specific amount of storage but do not have any visible members, assume there is exactly one hidden variable and print it. Do the values for the different classes correlate in any way and/or are they meaningful at all? The aforementioned worksheet might help you.
- Focus on the classes which have at least one virtual attribute, now. Compute the differences of the hidden member variables of an inherited class and its base class. Do you have an idea what might influence this difference? Check the declaration of the classes.
- Assume that the hidden value points to some sort of table. Assume further that its total size is the difference in bytes and each element is equal to the size of one pointer on your system. Given this information, print the table and look at its contents using the stub function which is part of the provided C++ file Do the entries match? Do they differ? Again, look at the different class declarations. Also have a look at the output of `objdump -d yourbinary`. Do you find some of the values of the hidden table? What is the purpose of this table and does it have a special name?
- CHALLENGE: Are you able to call the virtual method "challenge()" with information provided by this hidden table and `objdump`? Follow the hints in the respective stub function which is also provided in the C++ file. Does this low-level approach work for every method? Think about classes which have real member variables.

Compilation

For the compiled program to provide meaningful output optimization must be turned off. Further, it might make sense to include debug symbols for the purpose of analysis via debugger.