Virtual-C - a programming environment for teaching C in undergraduate programming courses

Dieter Pawelczak, Andrea Baumann
Faculty of Electrical Engineering and Computer Science
University of Bundeswehr Munich (UniBw M)
Neubiberg, Germany

***Abstract*—The C programming language plays an important role in the education of engineers especially in the field of embedded systems. On the other hand C is often a burden for students in the first year, as they have many difficulties in writing their own programs and the failure rates of course examinations are typically high. We have investigated different approaches at our faculty in the last years, how to enhance the students' capabilities in C programming and how to lower the failure rate of the C-programming course. Several concepts involved are: visualization of control and data flow, game programming and automated assessment tools with plagiarism detection. We have integrated some of these concepts into the programming environment Virtual-C IDE. This paper discusses the educational benefits of using the Virtual-C IDE for lectures, self-learning and as a platform for programming assignments and reports our first practical experiences.**

Keywords—computer science education, program visualization, programming environments, debugging, C

I. INTRODUCTION

The C programming language is taught at many technical faculties. Although this language is very closely linked to the workings of a machine, many students have difficulties learning it. These difficulties are wide spread and range from language specific problems to difficulties regarding programming concepts and problem solving in general [1].

We have experienced a high failure rate and a rising dropout rate of the C-programming course at our *Faculty of Electrical Engineering and Computer Science* (*UniBw M*, Munich) in the last years. In order to lower the failure rate and to enhance the students' capabilities in C programming we have taken different measures. These include from visualization of data and control flow, graphical programming, game programming, web-based quizzes and questionnaires to an automatic assessment tool with plagiarism detection.

These different approaches pushed the need for an integrated programming environment that includes all these features and can serve as a platform for the complete programming course, i.e. for demonstrations during the lectures, for self-learning, for preparations towards the examination and for accompanying programming assignments. In the last three years we have been developing the *Virtual-C IDE* (VIDE)[1]. This paper discusses the educational benefits of VIDE and presents our first practical experiences from using it in the course.

II. REVIEW OF RELATED WORK

To counter the problems in programming, a variety of programming environments have been developed for learning and teaching programming [2]. One of the most known is *BlueJ* [3] with the focus on object-oriented programming (OOP). Other studies discuss languages for introductory courses [4, 5]. Both topics – OOP and the choice of an introductory language – are out of scope of this paper: first, the C-programming course discussed here follows an introductory course based on the Java language; and second, the C programming language is required due to a high focus on embedded systems in our curriculum.

With respect to procedural programming common discussed difficulties are: control structures, functions/ subroutines, recursion, primitive and abstract types [1, 6]. More specific topics are language related problems such as buffer overrun and memory leaks [7]. We also found severe problems regarding uninitialized data as well as pointers without proper memory allocation. VIDE provides visualization means for all of these topics except for buffer overruns, which currently can be watch indirectly only.

As prior studies showed, a visualization or animation of programming concepts without interaction of the student has only a small impact on the student's understanding [8]. It can be useful to emphasize facts or to lighten the mood during lectures. An important fact is, that the lecturer should teach students how to work with the visualization tool. In addition, students use preferable one integrated visualization tool instead of several different ones [9]. A good example of several visualizations integrated in one tool is *COAC#*, that supports multiple visualizations for different languages in one tool [10]. Unfortunately *COAC#* is not yet integrated in a programming environment, so that a student still has to use different tools for programming and visualization. The software visualization *PROVIT* uses a similar approach as VIDE: it visualizes local variables on the call stack and shows the interaction of pointers

[1] Online available at https://sites.google.com/site/virtualcide/

[11]. However *PROVIT* offers no such extensive expansion options such as VIDE.

Debugging plays an important role to understand the workflow of algorithms as well as to find bugs in a program. Surprisingly, as [1] shows, debugging is rarely in the focus of educational programming environments. As we use debugging a lot during our courses, debugging became the central concept of VIDE, on which most of its features are based.

Although most studies reveal, that using programming environments does not cause major difficulties for students [6, 12], a simple to use environment allows focusing more on programming than on configuration and handling of the tool. A special benefit of VIDE is – as discussed before – the integration of visualization, debugging, editor and build system into one integrated tool.

III. Using the Virtual-C IDE for Lecture and Self-Learning

A. Usability

VIDE was initially designed for live-coding (compare e.g. [13]) and visualizations of control and data flow during the lectures and for students to work with these visualizations. Although professional IDEs can be configured in such a way, that fonts are large enough to be readable in the auditorium, these configurations are time consuming as they differ very much from the regular ones. An important design criterion therefore was the simple scalability of the dialogs to be usable with different beamer screen sizes.

Another design criterion was the one button usage: selecting any of the debug buttons will automatically compile and link the program's source in the active editor window and start the debugger. There is no need of a project definition, as demonstrations during lecture typically consists of single C files with tiny examples. The easy handling of the IDE simplifies live-coding in the lectures and allows students to concentrate more on the coding than on operating the IDE.

B. Virtual Machine

Programming and debugging a real machine is a very complex task. A pointer in a 64-bit process is often displayed with 16 hexadecimal digits; comparing two pointers is therefore difficult for the user even if they are identical. On the other hand simple examples rarely need much memory. Therefore VIDE uses a 32-bit virtual machine (VM) with a very simple memory layout (code, constants, data, stack, heap, hardware). User memory is always below 16 G; the debugger therefore presents pointers with typically 6 hexadecimal digits.

C. Visualization of data

During stepwise or slow motion execution all valid symbols of the process are visualized as memory blocks with a coloring scheme referring to the memory type (e.g. invalid, constant, stack, heap etc.). The smallest memory block refers to a single byte. Memory contents can either be displayed as byte content (hexadecimal) or as type specific content, i.e. integers like e.g. `short`, `int` as decimal, floating point values like e.g. `float`, `double` in exponential notation and `char` as characters. Pointers are literally drawn as arrows pointing to the referencing memory blocks, compare Fig. 1. Thus the user can detect memory changes during debugging both by content and color.

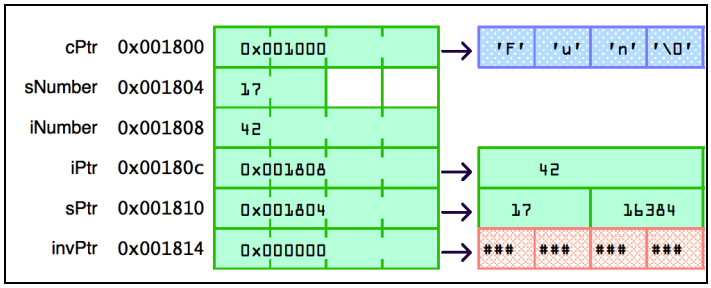

Fig. 1. Visualization of variables as memory blocks

After building a program, the IDE displays an overview of the process's memory layout. When the user starts the debugger, the content of all memory segments containing data are shown: constant data (mainly string literals), data for global variables, the call stack of each thread and the memory on the heap allocated by the process. As we avoid global variables, a student usually can see all variable contents in the visualization of the call stack. In case a function returns, the IDE will keep the stack contents of this function as long as possible; all these memory blocks are drawn in red to emphasize, that local variables become invalid. A common pitfall in C is for instance returning a pointer to a local variable, which is no longer valid.

The visualization of symbols as memory blocks is integrated in the IDE and per default available during debugging. The visualization helps students to understand concepts of different memory locations in a machine, different primitive types, simple pointers as well as passing parameters to functions by value or by reference. Although the IDE supports a classical watch window as well, the call stack is typically sufficient for watching the contents of local variables; furthermore, the call stack does not only show the actual value of a variable, it also displays the contents of identically named variables in different instances of functions, i.e. a common scenario in recursive functions. On the other hand, an important application of the watch window is the visibility of variables.

D. Visualization of advanced data structures

In order to create data visualization for advanced topics, e.g. for sorting algorithms or abstract data types, the IDE is extensible. As a bachelor thesis, J. Wonneberger wrote an extension to visualize binary trees and single-linked lists [14]. Fig. 2 shows his visualization in action: a representation of an unbalanced binary tree (2a) and the modified tree after balancing (2b). As the view is updated with every execution step, a student can watch the tree-modifications during insertion and balancing. Such visualizations do not only help to understand the data flow of algorithms, they also allow students to debug their own tree or list implementation: a cycle in a tree, a mismatched child node or an unintentionally shortened list is immediately shown in the visualization. The extension shown in Fig. 2 was written in C++. As accessing the internal data of the compiler and the VM requires a quite complex interface, the demand for a much simpler interface arose.

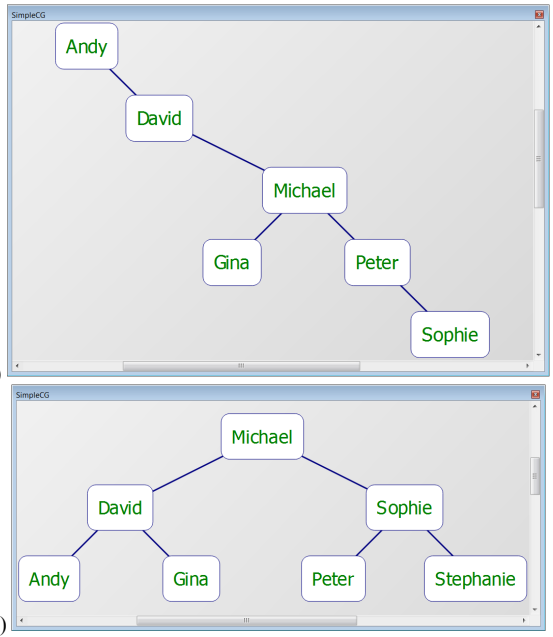

Fig. 2. Visualization of binary trees (unbalanced and balanced insert) – screenshots created with the IDE extension described in [14]

An easy to use plug-in mechanism has been established: students can write individual extensions to the IDE using *JavaScript* (JS) and access data from VIDE via JS objects. The graphical presentation can be done e.g. by a HTML5 canvas element.

```
<html><body>

<h1>Local Variables</h1>

<p id = "insert">-</p>

<script>

function OnBreaked() {

    var elem = document.getElementById("insert");
    elem.innerHTML = "-"; // clear child nodes

    // get current functions on stack
    var functions = VCThread.getFunctions();

    // get local variables of active stack frame
    var vars = VCThread.getLocalVariables(
                        functions.length-1);

    // print local variables
    for (var i = 0; i < vars.length; i++) {

        // get and print type
        elem.appendChild(document.createTextNode(
               VCThread.getVariableType(vars[i]) + " "));

        // get and print name
        elem.appendChild(document.createTextNode(
               vars[i] + " = "));

        // get and print content
        elem.appendChild(document.createTextNode(
               VCThread.getVariableValue(vars[i]) + ","));
    }
}
</script></body></html>
```

Fig. 3. A a simple plug-in to print the contents of local variables

An excerpt of a simple plug-in is presented in Fig. 3: it prints the type, name and content of each local variable. The realization is done in the `OnBreak()` method, which is called, whenever the process is interrupted. It queries from the current thread all functions of the call stack and then iterates over all local variables in the active stack frame. This simple JS interface allows students to implement advanced data visualizations without detailed knowledge of the VM.

E. Visualization of control flow

The visualization of a program's control flow is especially important regarding loops, conditional instructions and recursive functions. A typical pitfall for instance is the value of a counter-variable in a for-loop after the loop; or respectively, the loop condition required to get a certain number of iterations in different kind of loops. Fig. 4 shows an active debug session with the next instruction highlighted. A simple animation of this example can be achieved by executing the program in *slow motion*: VIDE runs the program step-wise, highlights the active line in the editor and prints the call stack, variables or memory contents in the watch or memory dump window respective.

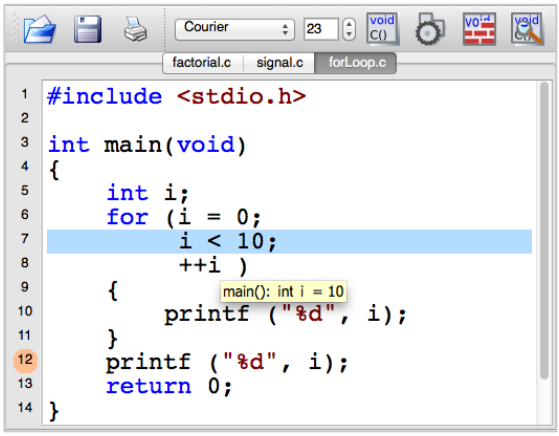

Fig. 4. Visualization of the control flow in the debugger with tooltips for variable contents

Visualization of the call stack is, as described in the previous section, especially important when debugging recursive functions. A work-in-progress is a plugin to additionally display the flow trace. Fig. 5 presents a prototype of the plugin during debugging a recursive calculation of the factorial: The plugin traces the call hierarchy and evaluates each line in the source code. The concrete contents of a parameters or variables are appended to the original code line in square brackets, e.g. `n[←3]` (3 is the current value of *n*). Each function call is intended and enclosed in two arrows (-> *call* and *return* <-). It can be collapsed, i.e. showing *call* (with concrete parameters) and *return* (with the evaluation result) only, to make the structure of the diagram clearer. Fig. 5 shows the complete flow trace of the sample program for a better understanding of the overall concept. Actually the flow trace prints only steps up to the currently debugged line (e.g. here: `return 1`), thus the following greyed-out lines would not yet be visible.

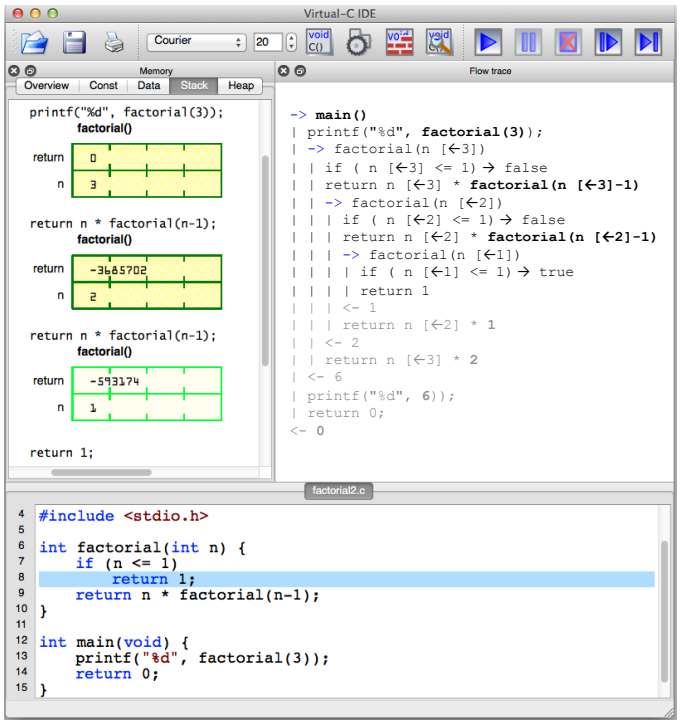

Fig. 5. Visualizating the flow trace of a recursive function (plug-in)

IV. Using the IDE for Programming Assignments

A. Simplified Usage

The tremendous options and dialogs of a professional software development system might overwhelm students and distract them from their programming assignments [15]. *VIDE* in contrary requires no configuration, provides a program skeleton at start up and a single button (*run* or *step*) simplifies the handling of compiler, linker & debugger.

B. Graphics-/ game programming

Programming assignments with graphical output enlarge motivation of students [16]. Some programming concepts, students feel hard to grasp, can easily be explained using graphics. A simple example is a visualization of numeric faults of floating point arithmetic in line drawing algorithms. Graphics programming is integrated in the IDE using the *Simple DirectMedia Library* (SDL). This platform independent library supports graphics and events like, e.g. mouse or keyboard actions [17]. A project skeleton to plot images and handle mouse events allows students to implement simple games like e.g. *Pairs* or *Roulette*.

C. Exercise assistant and electronic questionnaires

A major feature of the VIDE is an embedded test framework (TF), that allows functional and performance tests. Students can stepwise solve programming exercises and check if their solution fulfills the requested requirements. The teacher can define a questionnaire-like checklist in HTML with additional test files written in C. Fig. 6 presents an example of such a checklist. Although in general an arbitrary layout can be used, we typically designed the checklist as a table, in which each line represents a certain task, the student has to fulfill and submit the results for the automated tests. A task can be a question (text or multiple choice input), a complete C program or any other user interaction with the IDE. In case the user selects the submit button, a JS method is invoked. This method can perform checks within the checklist, as e.g. user input in a form, a multiple choice test, or run a functional test on the C program of the user, see Fig. 6.

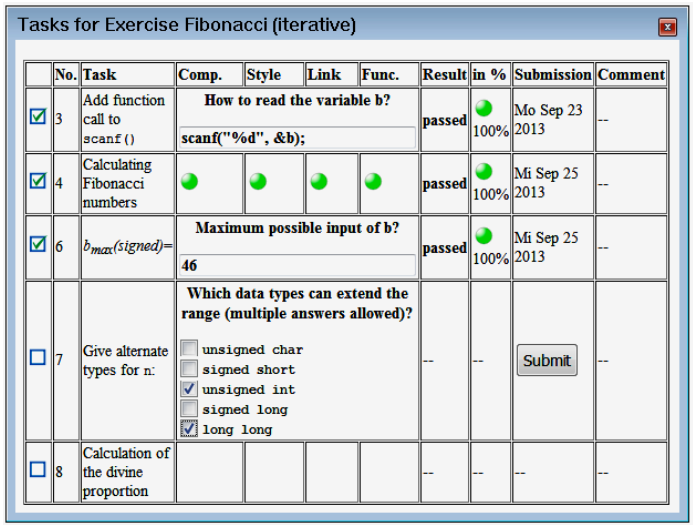

Fig. 6. Example of a questionnaire with integrated functional tests designed as a checklist

In case of a functional test, VIDE will first run static tests and report these results to the user: compilation of the program (*Comp.*), check on the coding style of the source file (*Style*) and link the user's program with the test file (*Link*). As Fig. 6 shows, the results of each step are displayed in a traffic light scheme. A green light reports success. In case of an error, e.g. the program does not compile, the submission is aborted and the user has to work on these errors before re-submission. After a successful build, the functional test is invoked (*Func.*). In case all steps passed and the overall result is above a given threshold, the task is passed and the overall result for the task is calculated and reported with a time stamp, compare Fig. 6.

Fig. 7 lists a simple example of a functional test, e.g. to check, if a student's program calculates the Fibonacci-number for the given input 8 correctly. The test framework provides several extensions to the C standard including access to statistic data from the compiler (static information). This data can be queried per function or for the overall program: for example the call `_queryIntAttributes("floatOps")` returns the total number of floating point operations of the program, whereas the attribute `main_floatOps` represents the count of float operations for the function `main()`. Other static information is for example:

- number of integer operations
- count of function calls
- maximum depth of loops and count of loops
- contents of defines
- values of enumeration types.

The TF executes the function `_mopcheck()`, which returns a number between 0 and 100, reporting the functionality as a percentage from poor to perfect. This function typically invokes functions of the student's program with a given input and checks the resulting output; the example in Fig. 7 calls the function `main()`. During the test run, the VM provides additional dynamic information (e.g. count of instructions, exceptions, stack and heap usage).

```
#include <mopvmex.h>

extern int main(void);

int _mopcheck(void){

    char result[512]; /* holds the output */

    /* static tests, e.g. number of float operations */
    if (_queryIntAttributes("floatOps") > 0) {

       _printErrorToOutput("Floating points are "
                           "not allowed!");

       return 0;
    }

    /* set test conditions: here redirecting stdin/stdout */
    _redirectStdout(result, sizeof(result));
    _redirectStdin("8\n", 2);

    /* limit CPU instructions, enable exception handler */
    _setExecutionLimit(10000);

    /* dynamic test: call the user program */
    if (main() != 0)
       _printWarningToOutput("main() should "
                             "return EXIT_SUCCESS.");

    if (_printFault("main() does not terminate.",
                    "unhandled exception."))
       return 0;

    /* check output of user program */
    if (!_containsRegEx(result, "\\b21\\b")) {
       _printErrorToOutput("main() does not print "
                           "the right Fibonacci-"
                           "Number for input 8.");

       return 5; /* at least no crash */

    }

    return 100; /* success, return 100% */

}
```

Fig. 7. A simple test program

To circumvent a deadlock or a crash of the test itself, when calling the student's code, the test specifies the maximum number of machine instructions (`_SetExecutionLimit`). In case this number exceeds or an exception occurs, the VM will continue execution in the stack frame, which called `_SetExecutionLimit`; thus continuing the test code as if the student's code had properly finished. The test program then can check the condition, for example querying the number of instructions or the kind of exception. The function `_printFault()` combines both checks and prints the corresponding error messages. The user in- and output can be redirected from/respectively to a string and – for convenience – a regular expression can be tested on a string, which simplifies testing the program's output.

In addition to the TF, the new plug-in mechanism (see Sec. III.*D*) allows access to the same static and dynamic information from JS. This offers a tremendous variety of options how to design the questionnaire: questions based on the student's solution can be generated dynamically. While the checklist in Fig. 6 shows a static question related to the data type ("which data types can extend the range"), querying the actual type used in the student's program allows modifying the question dynamically: imagine, a student already used an `unsigned long long int`, the question could be for instance: does the type `double` extend the range of possible Fibonacci-numbers or not?

D. Automated assesment system and plagiarism detection

Checklists and functional tests for exercises as described in Sec. IV.*C* can either be done standalone (i.e. loading the exercise from a local file folder) or integrated into a course's assessment system. Therefor VIDE provides a web interface, which can retrieve exercise descriptions from a web server and upload the filled-out checklists back to the server. In the C programming course at our faculty, a utility program called *MopClient* manages the student's authentication as well as a checklist on the overall results of the programming assignments. This tool invokes VIDE with the proper checklist for each assignment, compare Fig. 8.

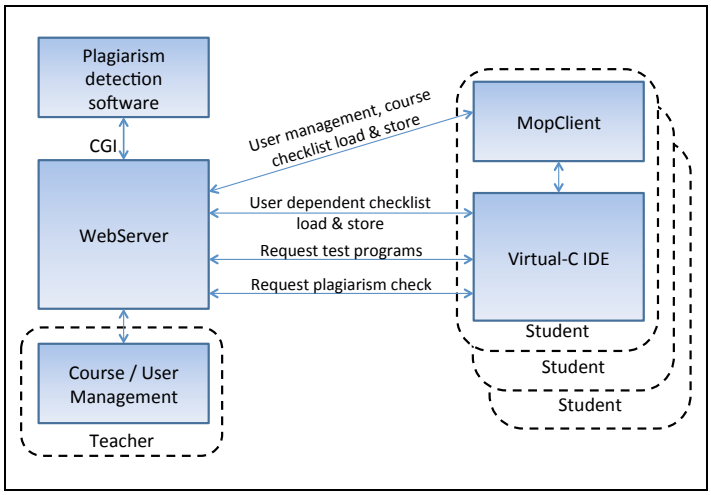

Fig. 8. System for automated assesment including plagiarism detection

For the programming assignment, VIDE requests test programs (as described in Sec. IV.*C*) for functional tests from the web server. Each completed task (either a pass or a fail) is uploaded to the web server in order to log the progress of the student and to store the individual state of the programming assignment. The teacher uses the course & user management system to specify the tasks per assignment, to define the number of allowed repetitions in case a submission is erroneous or incomplete, to adapt individual assignments and to monitor the students' achievements and the overall course results.

A plagiarism detection system can be integrated to prevent unwanted collaboration: the submission of a programming assignment is tested against all submissions from the fellow students. Exceeds the similarity of two submissions a given limit, a submission is rejected. As the program sizes of the

assignments are quite small (source lines of code are typically between 40 to 200) the limit is about 80%. For details on the plagiarism detection system see [18].

Although the assessment system would allow submissions at any time, we restrict submissions to class time and to our internal lab network. So students can prepare their programming assignment at home, but have to come into the lab for their submission. This allows our course instructors to help students with their submissions and to discuss individual programming difficulties.

V. Evaluation of the Virtual-C IDE

In the years 2011 and before, a commercial IDE had been used in the C-programming course. VIDE had been used for demonstrations during lecture in the spring course 2012 next to the professional IDE, which had been used for programming assignments and exercises. Although the courses evaluation revealed, that students like the live-coding and these demonstrations, only a view students downloaded VIDE and run the examples on their own. As discussed in Sec. II the effect of these visualizations was negligible: the failure rate slightly increased and the number of students with above 75% of all points even decreased a little bit, compare Fig. 10. The usage of two different programming environments during the lecture might be one reason, that students downloaded less of the programming examples, which had been shown during the course. Fig. 9 shows the download rate of 8 typical programming examples like e.g. integer arithmetic, pointers, strings, dynamical data structures and file handling, etc.

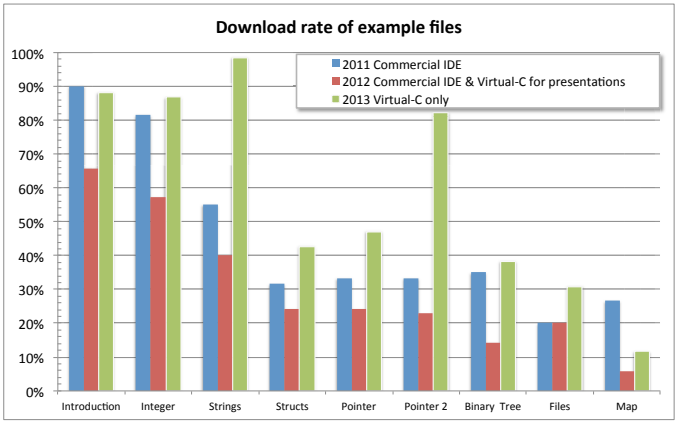

Fig. 9. Download rate of example projects/ files given during the lectures in the last three years

In the spring course 2013 VIDE had been used for the complete course, i.e. during the lectures, for exercises and for the programming assignments. The failure rate dropped down more than 15 % while 20 % more students reached 75 % and above points in the examination compared to the previous years, see Fig. 10. All examples shown in the lecture had been provided as single C-files except the last two examples, which used multiple files and were therefore zipped folders. The average download rate was about 29 % respectively 13 % above the rates for the last two years. Although the download rate does not give sound proof of the VIDE's acceptance, it can certainly serve as an indicator. As the student's evaluation of the course showed, most students downloaded the IDE on their private computer and most students debugged at least at one example.

The formal and pedantic functional tests of the automated assessment system required, that students put more time into their programming assignments at least with respect to fixing errors: the functional tests for the assignments revealed more programming faults, like e.g. invalid pointer usage, memory leaks or uninitialized variables, compared to manual checking in the previous years; course instructors didn't had the time to scrutinize that detailed source codes of each student; especially as the number of students is high compared to a dissatisfactory low count of course instructors. The automated assessment system helped instructors to work with students on programming issues more than on administrative tasks.

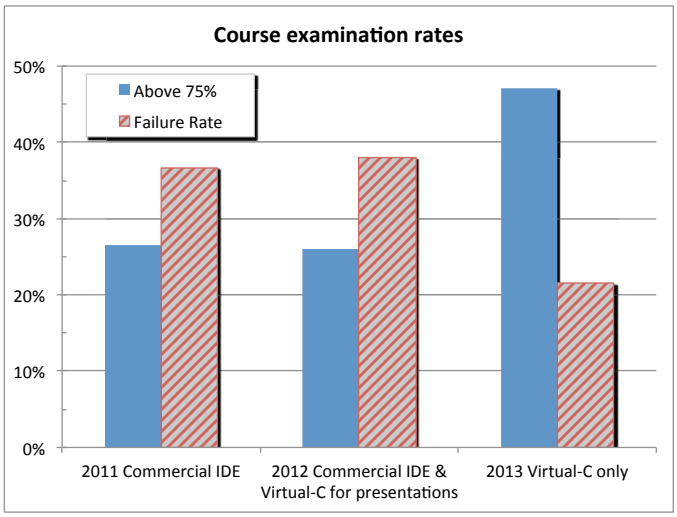

Fig. 10. Examination results of the C-programming course in the last three years

Two surprising results of the students' evaluation of the 2013 course are:

- students stated a slightly lower workload compared to the years before. The impressions of the course instructors were contrary: students seemed to work harder for the 2013 course. A higher motivation of the students competing with the assessment system might explain this difference
- and students wished electronic submission to be available outside the class time.

Although students appreciated the help of the instructors very much, some students preferred an anonymous submission. Such an "open round-the-clock" submission system would certainly encourage students' academic freedom, but also would counteract our pedagogic approach to help and discuss programming difficulties with students in class time.

VI. Conclusion and Outlook

We successfully introduced the *Virtual-C IDE* (VIDE) as a standalone programming environment for the complete C-programming course (including live-coding in the lectures, for the accompanying programming assignments and self-learning for the course examination) at our university. Although many different factors may influence the examination results, the failure rate significantly dropped more than 15 % compared to previous years, see Fig. 10. The students' evaluation of the course revealed, that many students liked the game-like approach of the checklists for exercises and automated assessment. Certainly, the electronic exercises and the simple handling of the debugger, compiler and visualization tools brought more students to actively work with the programming environment as in the years before. Another benefit of VIDE are the pedantic functional tests, which required, that students had to work harder on their assignments, as typical programming faults like, e.g. uninitialized data or memory leaks, were rejected in submission. Last but not least the plagiarism detection prevented unwanted collaboration and required each student to prepare an individual solution. Although some solutions resulted from teamwork, these had to be elaborated before submission. Students surprisingly stated a lower workload compared to the previous years, although the impression of the course instructors was that students spent more time for the course.

With respect to the future development of the *Virtual-C IDE*, we will extend the number of questionnaires for smaller and shorter exercises and plan to improve the static analytics of the compiler to give better feedback on typical coding mistakes. As VIDE is based on the open source library Qt[2] a current work in progress is a port to Android based mobile platforms.

Our future research focus is to further combine the visualization of complex data structures, like e.g. binary trees, with the presentation of the control flow: A challenging approach is to store the history of a program execution for both its data and control flow in order to allow students to navigate forward and backward in the visualization. We expect students to get a better understanding of their own programs, to learn from their mistakes and therefore enhance their programming capabilities.

Acknowledgment

We like to thank the students that contributed to the *Virtual-C IDE*: D. Schmudde, M. Huettner and J. Wonneberger and the instructors in the lab for their feedback: T. Groetzbach and M. Schneider.

[2] see http://qt-project.org

References

[1] M. Hertz, S. M. Ford, "Investigating factors of student learning in introductory courses". In Proc. of the 44th ACM technical symposium on Computer science education *(SIGCSE '13),* New York, NY, USA, 2013, pp. 195-200

[2] C. Kelleher, R. Pausch, "Lowering the barriers to programming: a taxonomy of programming environments and languages for novice programmers". ACM Computing Surveys, Vol. 35, 2 (June 2005), pp. 83-137

[3] M. Kölling, B. Quig, A. Patterson, J. Rosenberg, "The BlueJ system and its pedagogy". J. Comput. Science Educ., Special Issue of Learning and Teaching Object Technology 12, 4, pp. 249–268.

[4] L.Mannila, and M. de Raad, "An objective comparison of languages for teaching introductory programming". Proc. of the 6th Baltic Sea conf. on Computing education research: Koli Calling 2006, New York, NY, USA, 2006, pp. 32-37.

[5] A. Dingle, and C. Zander, "Assessing the ripple effect of CS1 language choice". J. Comput. Sci. Coll. 16, 2 (October 2000), pp. 85-93

[6] E. Lahtinen, K. Ala-Mutka, and H.-M. Järvinen, "A study of the difficulties of novice programmers". SIGCSE Bull. 37, 3 (June 2005), pp. 14-18

[7] A. Allevato, S. H. Edwards, and M. A. Pérez-Quiñones, "Dereferee: exploring pointer mismanagement in student code". SIGCSE Bull. 41, 1 (March 2009), pp. 173-177

[8] C. Hundhausen, S. Douglas, and J. T. Stasko, "A meta-study of algorithm visualization effectiveness". J. of Visual Languages and Computing, 13 (3), 2002, pp. 259–290

[9] T. Naps, S. Cooper, B. Koldehofe, C. Leska, G. Rößling, W. Dann, A. Korhonen, L. Malmi, J. Rantakokko, R. J. Ross, J. Anderson, R. Fleischer, M. Kuittinen, and M. McNally, "Evaluating the educational impact of visualization". SIGCSE Bull. 35, 4 (June 2003), pp. 124-136

[10] M. A. Jakeline, "Learning computer programming with COAC#". FECS'13: The 9th Int. Conf. Frontiers in Education: Computer Science and Computer Engineering, July 22-25, 2013, Las Vegas, USA, pp. 160-165

[11] K. Matsumura, S. Daisukey, A. He, "A C language programming education support system based on Software Visualization". Joint Conferences on Pervasive Computing (JCPC), 2009, pp. 9-14

[12] S. Xinogalos, "Programming techniques and environments in a technology management department". In Proc. of the 5th Balkan Conf. in Informatics (BCI '12). ACM, New York, USA, pp. 136-141

[13] M. J. Rubin, "The effectiveness of live-coding to teach introductory programming". Proc. of the 44th ACM technical symposium on Computer science education (SIGCSE '13)., New York, NY, USA, 2013, pp. 651-656

[14] J. U. Wonneberger, "Developing an application for a graphical view of dynamic data structures". Bachelor thesis, UniBw M (in German): „Entwicklung einer Anwendung zur graphischen Darstellung dynamischer Datenstrukturen", 2012

[15] E. Dillon, M. Anderson, and M. Brown, "Comparing feature assistance between programming environments and their "effect" on novice programmers". J. Comput. Sci. Coll. 27, 5 (May 2012), pp. 69-77

[16] A. Slaby, and E. Milkova, "Computer graphics as a way of improvement programming skills". ITI 2006, 28th Int. Conf. Information Technology Interfaces, June 19-22, 2006, Cavtat, Croatia, pp. 295-300

[17] B. Pendleton, "Game programming with the sdl". *Linux J.* 2003, 110 (June 2003)

[18] D. Pawelczak, "Online detection of source-code plagiarism in undergraduate programming courses". FECS'13: The 9th Int. Conf. Frontiers in Education: Computer Science and Computer Engineering, July 22-25, 2013, Las Vegas, USA, pp. 57-63