CLIP, a Command Line InterPreter for a subset of C++

Harri Luoma **Essi Lahtinen** **Hannu-Matti Järvinen**

Tampere University of Technology
Institute of Software Systems,
Tampere, Finland.
harri.luoma@tut.fi, essi.lahtinen@tut.fi, hannu-matti.jarvinen@tut.fi

Abstract

C++ is not the best choice for a first programming language, but if it is used, the learning circumstances need to be as easy as possible. We have developed a pedagogically designed interpreter for this purpose. Our hypothesis is that an interpreter is easier than a compiler for a novice programmer to use. We do not want students to use language properties they do not yet fully understand, so our approach is imperative-first. In addition, when using an interpreter, learning concepts such as libraries can be postponed until later in the course.

C++ is a complex language and most of its language features are not needed by a novice programmer. Therefore we have simplified the language to a subset of C++ that we call C--. For instance, classes have been omitted. We have also put a lot of effort into producing clear, informative error messages, something that is made possible by of the simpler programming language.

This article introduces some other C/C++ interpreters, their evaluation, and the description of our interpreter called CLIP. So far CLIP has not been used by students, so its evaluation is left as future work.

Keywords: interpreter, C++, novice programmers, education

1 Introduction

To make the start of learning programming as easy as possible for our students, we have developed a tailored interpreter to address the needs of novice programmers. For external reasons, we have to use C++ as the programming language for our first programming course. However, to keep the concepts needed in the first programs as simple as possible, our teaching approach is imperative-first. Since C++ is such a complex programming language, we have chosen a subset of C++ to use in the beginning.

Our hypothesis is that using an interpreter is simpler than using a compiler. One of the reasons for this is that the feedback is received faster from an interpreter than from a compiler. When using a compiler, there is a lot of overhead in writing the first complete program. In C++, the student has to write *include* directives, *using* statements and a *main* function without knowing what they actually do. These are often experienced as mystical phrases or 'spells' since their meaning is not yet understood. Using a compiler also requires the novice students to learn how to edit the program, save it, compile it, and, finally, run it. All these form a set of obstacles to the comprehension of the whole programming process. The use of an interpreter can help postpone the learning of these difficult phases.

In addition to eliminating the need for these 'spells', excluding the complicated language features, and simplifying the programming process, the purpose of our interpreter is to be able to give clear error messages in the students' native language.

In this article, we introduce the CLIP C-- interpreter. We also present some reasons why the tool has been implemented and explain how it is going to be used. The article is organized as follows: In Section 2 we present the background of CLIP. Section 3 discusses novice programmer behaviour in general and Section 4 presents some other C++ interpreters for comparison. Section 5 describes CLIP and its programming language. Section 6 contains some evaluation, and Section 7 the conclusions.

2 Background

In teaching programming, the intention is to teach not just a programming language and its concepts but problem solving and programming knowledge in general. However, a real programming language is needed for practical training. Some teachers have tried a syntax-free approach to programming (Fincher 1999) where instead of a programming language, some kind of formal structures are used to present programs. This has not been well received, mainly because of the lack of motivation of students. They feel that they are not learning 'real programming' if no real-world programming language is used in teaching.

To address this problem, there are many languages designed especially for teaching novice programmers. Their approaches vary widely (Kelleher & Pausch 2005). One of the approaches to ease learning is to simplify an existing language, which normally leads to a separate language for teaching purposes.

Unfortunately, the teacher is often not free to base the choice of the first programming language on pedagogical criteria alone. For instance, the choice might be influenced by the university or the requirements of industry. Thus the teachers' duty is often to make it as easy as possible for the students to learn the chosen language. Since in our context the pressure to use C++ is quite high, we have chosen to simplify C++ to get the benefits of a simple language.

3 Behaviour of novice programmers

Winslow (1996) has listed differences between expert and novice programmers, one of which is that the

novices do not have routines for the lower-level programming activities the same way the experts do. Thus, in the beginning it is essential to make the use of the programming tools as simple as possible, leaving the novice with more time and energy for learning the higher-level activities. This is the main reason why we decided to use an interpreter instead of a compiler.

Jadud (2006) has followed the edit-compile-cycle of novice programmers. He reports that students often get stuck with an error reported by the compiler and have problems in proceeding further. Surprisingly, a novice programmer can take hours to correct a simple syntax error that would require a couple of seconds from an experienced programmer. Perkins et al. (1989) report similar findings. They identified groups of novice programmers that behave in different ways when using a compiler: the stoppers, the movers, and the tinkerers. The stoppers tend to give up when they face a problematic situation, while the movers try to explore the problem and often finally solve it. The tinkerers can also be called the fast movers since their strategy is to try all different possibilities. Unfortunately, they choose the changes in a random way, so this kind of learning is not effective. Jadud's study also reveals that tinkering is a common strategy among novice programmers.

To prevent students from tinkering or stopping, and generally wasting their time with practical problems, the compiler's error messages should be made as informative and clear as possible. If the error messages directly give the first hint on how to proceed in correcting the program, the student can solve more problems without help, and hence get more confidence in his or her programming skills. This has been our goal when implementing CLIP.

4 Existing C/C++ interpreters

Before implementing CLIP we familiarized ourselves with some other C/C++ interpreters. Kölling (1999) discusses object-oriented programming languages and environments and gives a set of requirements for them. Most of these requirements also apply to imperative programming environments: ease of use, integrated tools, support for code reuse, learning support, group support, and availability. We have used these requirements to evaluate the following interpreters.

***Ch* (2007)** is a C/C++ interpreter that supports a superset of C with C++ classes. It extends the C language for scripting, numerical computing and 2D/3D plotting. It has a graphical interface called ChSciTE, where the code can be edited. We did not evaluate the commercial version, but found the free version unreliable.

***CINT* (2007)** is a command line C/C++ interpreter aimed at processing C/C++ scripts. The language interpreted by CINT is a hybrid of C and C++. The syntax is less strict than C++, which means that not everything that runs on CINT can be compiled with a normal C++ compiler. There are also some differences in template libraries. Some C++ code may not work with it or may even be run in a wrong way. The interpreter is not very intuitive to use and is not meant for novice programmers or programming education.

***UnderC* (2007)** is a fast C/C++ interpreter and is intended for programming education. It has a graphical editor, and it is working pretty well. There were some issues left in the GUI, and it has been in beta phase since 2003, so we do not expect it to be developed further.

IfNCP (Sankupellay & Subramanian 2005) is an interpreter for novice C programmers. It is not available free of cost so we did not have an opportunity to test it. It is a web-based interpreter with an interactive online learning package but it only covers five basic C programming concepts. C++ is not supported.

Unfortunately, none of these interpreters meets our requirements or the requirements set by Kölling (1999). In brief, Ch is not reliable enough, CINT is not simple to use, and UnderC and IfNCP do not meet the availability requirement.

We also want to be able to add graphical features to our interpreter. It would be quite difficult to integrate a visualization module with these existing interpreters, so that is another reason for implementing our own interpreter.

5 CLIP

The main ideas of CLIP are to support interpreter-based teaching and to give students better and clearer error messages in their native language. In addition to these, it should be possible to integrate a visualization module with the interpreter. This requirement has its roots in our visualization tool VIP (Virtanen et al. 2005), since we are not willing to maintain two separate interpreters.

C--, as we call the subset of C++ interpreted by CLIP, includes most of the basic structures of C++. We have selected the imperative paradigm for our elementary courses, which is a common approach when teaching C++ as the first programming language. Hence we have omitted some advanced features such as classes, namespaces, and exceptions. In addition, constructs that most often cause careless mistakes have been removed, or the interpreter deals with them more strictly.

This way, some of the problems of C++ can be diminished, if not avoided. We are aware that our changes do not make C-- an optimal language for teaching, but they do ease the task of the novice learner.

CLIP can be used in two modes, an interactive mode or a single-file mode. In the interactive mode, users write commands directly for the interpreter as they would be written inside the main function. The user can still define structs and functions normally. In the single-file mode, the user can give CLIP a single file, which is then interpreted and executed like any script. The command line interface of CLIP can be seen in Figure 1.

5.1 The supported properties

The features of C-- were dictated mainly by the requirements of the first programming course where we intend to use CLIP. Some features were left out because they were considered harmful for students.

The main features that are missing are following:

- *goto* statement
- classes (except *cin*, *cout*, *string*, *vector*)
- file streams and string streams
- bit operators (to prevent confusion between & and &&)
- exceptions
- namespaces

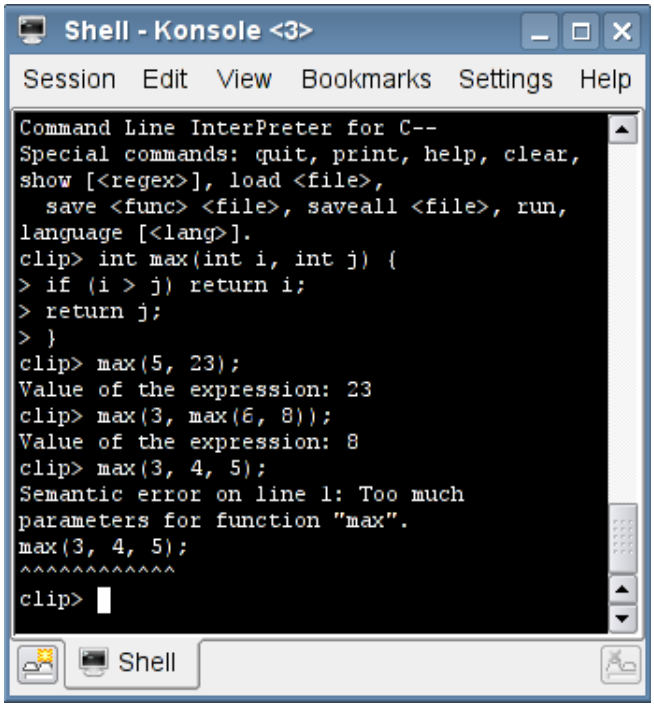

Figure 1: CLIP user interface

- templates (except vector)
- overloading of function names.

CLIP adds these features compared to a standard compiler:

- When an error is found from the code, the line that caused the error is shown.
- Error messages are localizable to student's native language.
- Contents of the symbol table are shown in runtime.
- We introduce *constraints*, which mean that the teacher can turn some features on or off depending on the topic to be taught.
- There are commands to save and load functions to and from external files.

5.2 Errors, warnings and other messages

There are eight different kinds of message: syntax, lexical, semantic, runtime, other errors, warnings, info messages, and constraints.

When an error occurs, it is shown instantly to the user, with no attempt to read the source code further. Further reading is usually needless because the later errors are often caused by the first error and do not clarify the situation at all. In fact, they may confuse the student and lengthen the error message output in vain. This approach has been proved to be viable in BlueJ (Kölling et al. 1999).

The constraints are a way to disable some C++ features from the interpreter. When disabled, the use of a feature will only produce an error message. At preset the following constraints can be enabled:

- switch statement
- pointers
- altering the for loop variable inside the for loop
- subroutines that have both non-const parameters and a return value (This constraint forces students to make a clear choice between a function and a procedure.)
- global variables
- arrays and vectors as members of a struct (This makes students use arrays of structs instead of a single struct with arrays in it.)

With the constraints, teacher can, for example, disable pointers when references are taught so that students cannot use pointers by mistake.

The interpreter also supports the following runtime error messages:

- divide by zero
- too deep recursion
- infinite loop
- reference through a null pointer
- index or pointer out of bounds
- use of uninitialized variables.

The output of CLIP is automatically split to fit the lines of the console. Individual words will not be split over multiple lines unless it is unavoidable.

5.3 CLIP compared to a normal compiler

The error messages given by a regular compiler, such as g++, are sometimes next to impossible to understand. This is often a consequence of the use of template libraries, but a missing semicolon in a vulnerable place can also cause very confusing error messages. The language supported by CLIP is much more constricted than C++, which makes it easier to check the syntax and to form clearer error messages.

In the next example we compare CLIP and the GNU C++ compiler (g++). We gave this piece of code to both of them:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> i;
  cout << i;
  return EXIT_SUCCESS;
}
```

The g++ compiler gave an error that was about 80 lines long and started with the following lines:

```
example.cc: In function 'int main()':
example.cc:6: error: no match for 'operator<<' in
'std::cout << i'
/share/gcc/gcc-4.1.1/lib/gcc/i686-pc-linux-gnu/4.1.1/
../../../../include/c++/4.1.1/bits/ ostream.tcc:67:
note: candidates are: std::basic_ostream<_CharT,
_Traits>& std::basic_ostream<_CharT, _Traits>::
operator<<(std::basic_ostream <_CharT, _Traits>& (*)
(std::basic_ostream<_CharT, _Traits>&))
[with _CharT = char, _Traits = std::char_traits<char>]
```

CLIP gave the following error

```
Semantic error on line 6: Can't print type
"vector of int" to cout.
  cout << i;
          ^
```

Clearly the error message of CLIP is easier for a novice programmer to understand.

In addition, when using CLIP, students would only need to write the following two lines to test the example code:

```
  vector<int> i;
  cout << i;
```

No struggling with namespaces or the main function is needed.

6 Evaluation of the tool

Since the interpreters discussed in Section 4 did not meet our requirements, we had to write an interpreter of our own. Hence, it is not a surprise that CLIP clearly meets our requirements. Unfortunately, we have not yet been able to gather student feedback on it. Tentative use by some faculty members has given encouraging feedback, so we are quite optimistic.

As mentioned above, Kölling (1999) lists properties of programming environments for the object-oriented approach. Of the applicable properties for imperative programming, CLIP does not provide group support at all, since in the first programming course each student should learn all the basic features individually. We believe that the rest of the properties have been met at least at the basic level, but without actual student feedback they cannot be evaluated.

However, we want to point out that in our opinion, CLIP gives good learning support. The main properties to support this have already been given: clear and informative error messages and the fast response of an interpreter. Furthermore, CLIP logs all the code the students input, so we get some data to examine. This data is intended to be used for improving the tool and the lectures. It will not be used for assessing the students.

Although we already think that CLIP is useful for our current course, utilizing its full potential requires rewriting the lecture material. The basic idea of the new material is that there will be no segment of program code in the material if the student does not understand why. We expect to have the first version of the new lecture material available for the academic year 2008-2009.

7 Conclusions

Selection of the first programming language for teaching should be done mostly on pedagogical basis. However, when facing cruel reality, this cannot always be the case. Our compromise has been to define C--, a limited subset of C++, and to offer a friendly interpreter, CLIP.

In this stage it is hard to tell how successful CLIP will be. With the new lecture material and integration of a visualization module, we expect to get a system in which the learner only uses the properties he or she knows, and in which the visualization will help the students not only to understand the program behaviour but also to debug their code.

8 Acknowledgements

The Federation of Finnish Technology Industries 100th anniversary and Nokia Foundation have partly funded this work.

References

Ch (2007). A C/C++ interpreter, http://www.softintegration.com/ (referenced 18 October 2007).

CINT (2007). A C/C++ interpreter, http://root.cern.ch/twiki/bin/view/ROOT/CINT (referenced 18 October 2007).

Fincher, S. (1999). What are we doing when we teach programming?, *in* 'Proc. of the 29th ASEE/IEEE Frontiers in Education Conference', pp. 12a4–1–12a4–5.

Jadud, M. C. (2006). Methods and tools for exploring novice compilation behaviour, *in* 'ICER '06: Proceedings of the 2006 international workshop on Computing education research', ACM, New York, NY, USA, pp. 73–84.

Kelleher, C. & Pausch, R. (2005). 'Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers', *ACM Comput. Surv.* **37**(2), 83–137.

Kölling, M. (1999). 'The problem of teaching object-oriented programming, part ii: Environments', *Journal of Object-Oriented Programming* **11**(9), 6–12.

Kölling, M., Quig, B., Patterson, A. & Rosenberg, J. (1999). 'The BlueJ system and its pedagogy', *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology* **11**(4), 249–268.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. & Simmons, R. (1989). Conditions of learning in novice programmers, *in* 'Elliot Soloway and James C. Spohrer, Studying the Novice Programmer', pp. 261–279.

Sankupellay, M. & Subramanian, P. (2005). 'Teaching C programming with the aid of an interpreter - online interpreter for novice C programmer (IfNCP)', *Jurnal Teknologi* **11**, 33–44.

UnderC (2007). A C/C++ interpreter, http://home.mweb.co.za/sd/sdonovan/underc.html (referenced 18 October 2007).

Virtanen, A. T., Lahtinen, E. & Järvinen, H.-M. (2005). 'VIP, a visual interpreter for learning introductory programming with C++', *Proceedings of The Fifth Koli Calling Conference on Computer Science Education* pp. 125–130.

Winslow, L. E. (1996). 'Programming pedagogy – a psychological overview', *SIGCSE Bulletin* **28**(3).