Thetis: An ANSI C Programming Environment Designed for Introductory Use

Stephen N. Freund and Eric S. Roberts
Department of Computer Science
Stanford University

ABSTRACT
Commercially available compilers, particularly those used for languages like ANSI C that have extensive commercial applicability, are not well-suited to students in introductory computer science courses because they assume a level of sophistication that beginning students do not possess. To alleviate this problem at Stanford, we have developed the Thetis programming environment designed specifically for student use. The system consists of a C interpreter and associated user interface that provides students with simple and easily understood editing, debugging, and visualization capabilities. Reactions of students and instructors indicate that Thetis fulfills the goals we set out to accomplish and provides a significantly better learning environment for students in CS1/CS2.

1. INTRODUCTION

When the Stanford Department of Computer Science decided in 1991 to adopt ANSI C as the language of instruction for its CS1/CS2 sequence, most of the affected constituencies strongly supported the change. In particular, students saw the C-based material as more relevant—an attitude shared by many faculty members, both inside and outside the department, who felt that the change would prepare students more effectively for advanced work.

Ironically, the only people to express any significant reservations about the decision to adopt C were one or two of the lecturers who would be responsible for teaching the introductory courses. There was little question in anyone's mind that Pascal was becoming increasingly obsolete and needed to be replaced. Likewise, there was no doubt that students needed to learn both C and C++ earlier in the curriculum than they had in the past. The concerns were focused instead on whether the added complexity imposed by using C in the introductory course would make it harder for students to master the more important issues of problem solving and algorithmic design. The fear, as one lecturer expressed it, was that "students would end up spending more of their time debugging and less of their time learning."

As we have reported elsewhere [Roberts93, Roberts95a], the change to C has had a positive effect on the computer science program and that we have in fact achieved most of the benefits we anticipated. Even so, it is important to note that the concerns were not without foundation. When we evaluated the C-based course in a seminar on undergraduate teaching after its initial year, we determined that students were indeed spending a larger fraction of their time trying to track down relatively obscure errors in their code and that the student frustration level was often higher as a result.

On further examination, however, we made an important discovery: student frustration was less a function of the *language* than of the *programming environment*. The Pascal compiler we had abandoned was much easier to use than the C compiler we adopted for the revised course, particularly for beginning programmers. The Pascal compiler detected more errors, provided clearer diagnostics, and included an interactive debugger that was much better suited to student use. It was these factors—not a change in the syntax and structure of the programming language—that made the difference.

After extensive consideration, we decided that the best solution to the problem of inadequate tools was to develop a programming environment specifically designed to address the needs of introductory computer science students. That project—since christened Thetis after the sea (C) nymph from Greek mythology—was used on an experimental basis in winter quarter of 1995 and then adopted for the entire introductory course in the spring. This paper describes the Thetis project and our experiences using it at Stanford.

2. SHORTCOMINGS OF EXISTING COMPILERS

The idea that weaknesses in the programming environment complicate the learning process for beginning students is not a new one; such shortcomings have been recognized in earlier papers [Ruckert93, Schorsch95]. The underlying problem is that most commercial compilers—particularly for languages like C that cater to a large audience of programmers—are designed for experts rather than novices. As a result, most of these compilers are poorly suited to student use.

In our experience, commercial compilers suffer from the following problems when used in an introductory course:

• *The error messages generated by commercial compilers are often uninformative and sometimes misleading.* New programmers tend to make certain mistakes more often than others. For example, omitting a required semicolon or right curly brace is a very common error in C programs. Unfortunately, compilers sometimes respond to this error by reporting a seemingly unrelated syntax error several lines below the actual mistake, presumably because the parser does not detect the problem until that point in the source file. Expert programmers understand the problem and know where to look; novices are completely baffled.

• *Commercial C compilers offer little or no run-time checking that would help students identify bugs that are otherwise extremely hard to find.* Although misleading error messages are a source of early confusion, the greatest frustration for students comes from logic errors that the system never detects. In C, the problem is especially pernicious because compilers rarely generate code to detect common run-time errors such as selecting an element outside the bounds of an array or dereferencing an invalid pointer. Even though such run-time checking would be helpful to novices, experts are unwilling to pay the cost.

• *Interactive debuggers typically require students to understand advanced concepts before they are ready to assimilate them.* Debugging is further complicated by the fact that existing debuggers usually require a high level of sophistication. The commercial C compiler we use at Stanford, for example, displays string data as a hexadecimal character pointer—a representation that makes sense only if you understand how C represents string data internally. In our course, we introduce strings as an abstract type very early in the course, deferring the underlying representation until close to the end. While this strategy makes good pedagogical sense, the failure of the debugging tools to present string data in an understandable way severely restricts the utility of the debugger.

• *Commercial compilers offer a bewildering set of options and special features that are useless to the novice and occasionally cause consistency problems.* Because their products must offer advanced programmers all the resources available on the machine, commercial programming environments typically have complex user interfaces that overwhelm the beginner who has no use for the added power and flexibility. To the student, these special features represent additional complexity that limits understanding and self-confidence. The problem becomes more severe if the student inadvertently chooses incorrect or incompatible options that prohibit successful execution of the program.

The main problem facing producers of commercial language systems is that compiler technology makes it hard to provide the kind of feedback that beginning users need. By definition, a *compiler* generates machine instructions that are later run as an independent process. Because the compilation phase is separate from the execution phase, the run-time system and debugger have limited access to the information generated during compilation. By contrast, an *interpreter* combines these operations into a single phase in which the interpreter itself simulates the program execution. Interpreters therefore incur a significant penalty in execution efficiency that makes them inappropriate for writing production code. Introductory students, however, have no need to achieve maximum run-time efficiency and would gladly sacrifice some execution speed for greater simplicity and reduced debugging time.

3. THE DECISION TO BUILD A C INTERPRETER

Although it was certainly possible to design a C interpreter to address the shortcomings identified in the preceding section, it was not immediately clear that doing so made practical sense. As serious as the shortcomings are, the idea of writing a new C interpreter from scratch represents a rather ambitious solution strategy. Building any software system of this scale and complexity is a major project that should not be undertaken lightly, particularly if other avenues are available.

To make sure that we understood the dimensions of the problem, we spent an entire quarter of Stanford's CS298 course, "Seminar on teaching undergraduate computer science," evaluating the proposed interpreter project, both to refine its goals and to assess its feasibility. In the end, we decided to undertake the project for the following reasons:

• *There was no existing system that satisfied both our pedagogical goals and our hardware requirements.* Even though it is possible to buy commercial C interpreters, such as CenterLine's CodeCenter product [CenterLine95], that offer many of the features we seek, those systems do not run on the hardware available on campus, which is primarily the Apple Macintosh. The introductory CS courses at Stanford enroll over 1200 students per year, making it infeasible to contemplate a change of platform.

• *Our pedagogical approach depends on the use of libraries that are most effective when well integrated within the programming environment.* Our success in teaching C depends to a large extent on the use of the cslib libraries [Roberts93, Roberts95a], which temporarily mask much of C's complexity until students are better prepared to understand the underlying representation. To ensure that students are not confused by conflicting models, it is important for all components of the interpreter to take account of the cslib definitions. Because cslib was developed locally, commercial products would be unlikely to support such integration.

• *We wanted to explore several new ideas in the design of a pedagogically based programming environment.* In the process of developing a preliminary design, we identified several features that seemed useful as experiments. For example, we were interested in exploring the relationship between debugging and dynamic visualization. Exploring this design space required that we develop a research tool that we could extend and modify. Although much of this research remains in the future, the prospect of undertaking such research provided a strong incentive to implement the interpreter.

• *We had a team of students who were interested in taking on the project.* At Stanford, students—even at the undergraduate level—are more fully integrated into the computer science teaching program than at many institutions [Roberts95b]. The idea of building a C interpreter for use in the introductory classes captured their imaginations and enthusiasm more than any other project idea in recent memory. Many students contributed ideas to the project and sought to become more involved. More importantly, there was a core of highly motivated, extremely competent students who wanted to drive the design and implementation. The existence of that group of students meant that the project would "succeed" no matter how usable the final system turned out to be; if nothing else, the implementers would have gotten an enormous amount of experience from doing the project that would significantly enhance their own education.

4. AN OVERVIEW OF THETIS

. . . Thetis was a faithful mother to [Achilles], aiding him in all difficulties and watching over his interests from the first to the last.

— Edith Hamilton, *Mythology*

Thetis is a complete, integrated development environment that "aids students in their difficulties and watches over their interests" by providing a conceptually simple, yet powerful, environment in which to learn programming. In many respects, Thetis presents a conventional user interface: a student creates source files in editor windows, indicates what files make up the application by listing the contents in a project window, and then controls the execution of the program through a menu-driven interface similar to that used by any Macintosh application. The major differences are that Thetis

- Automatically incorporates the cslib and ANSI libraries, so that students need not be concerned with them
- Offers significantly enhanced error detection
- Provides a convenient debugger/visualizer that operates at a conceptual level appropriate to the students' understanding

The sections that follow expand on the most important new features Thetis provides.

Improved error reporting

In his discussion of the CAP utility for analyzing and reporting errors in Pascal programs [Schorch95], Tom Schorsch notes the difficulty students have comprehending the syntactic error messages generated by many compilers. His experiences with Pascal are similar to our own with commercial C compilers: commercial environments simply do not provide error messages understandable by novices.

The error messages reported by Thetis are designed to give the user much better feedback than the usual terse messages reported by compilers. For example, the lines

```
char c;

c = "a";
```

often generate errors like "type mismatch" or "assignment makes integer from pointer without cast." Neither message is particularly useful to the introductory student. The first gives no helpful information regarding the cause of the problem, and the second phrases the error in terms a student may not comprehend. Thetis provides a more suitable error message for a beginning programmer: "cannot assign a value of type **`string`** to a variable of type **`char`**." While this example may appear trivial, enhancing all error messages in a similar fashion, as we have done in Thetis, makes it much easier for novices to understand and learn from their mistakes.

Strengthened syntactic restrictions

One of the problems faced by instructors using C or C++ in an introductory course is that these languages regard as legal certain syntactic constructs that are almost always errors when written by students. The most common example is using the assignment operator = in place of the relational operator ==, as illustrated by the following line:

```
if (i = 0) . . .
```

In C, this statement is perfectly legal. The variable **`i`** is assigned the value 0, which is interpreted as **`FALSE`**. In contrast, Thetis flags this line as an error by default, because its effect is almost certainly not what the student intended. Other examples of constructs likely to be erroneous include using relational operators to compare strings and failing to include function prototypes.

The arguments for enhanced syntactic checking in Thetis are similar to those outlined for the syntactic restrictions of Educational C [Ruckert93]. When such checking is enabled—as it is for the introductory classes—students make fewer errors and can concentrate on developing better programming skills.

Run-time error detection

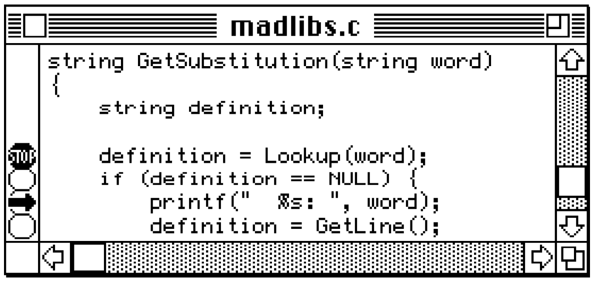
For introductory students, the most frustrating phase of debugging is finding the logic errors that persist after all syntactic errors have been corrected. C environments are traditionally not very useful when such errors arise, offering little feedback to help programmers find bugs in misbehaving code. To reduce this burden, the Thetis interpreter checks for the following run-time errors:

- Dividing by zero
- Using an uninitialized variable
- Dereferencing or freeing an invalid pointer
- Accessing an array out of its bounds
- Assigning an out-of-range value to an enumerated type
- Exiting a non-**`void`** function without returning a value
- Calling a bad function pointer
- Passing invalid arguments to a function

Debugging and visualization tools

Previous experience has shown that debuggers do not have the features needed to present a clear conceptual view of a running program to novices [Birch95]. By contrast, Thetis is designed to present information about the execution state of a program in a way that is consistent with the novice programmer's level of understanding.

To minimize the learning curve for new programmers, we have designed Thetis so that it builds on paradigms with which students are already familiar. The following screen diagram, for example, shows a Thetis editor window:

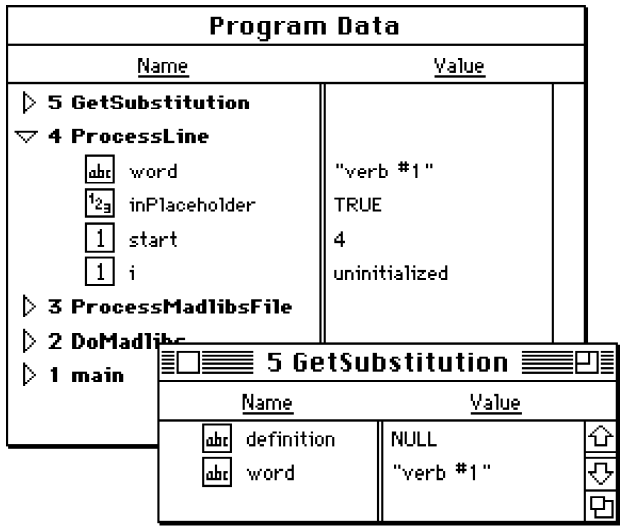


```
string GetSubstitution(string word)
{
    string definition;

    definition = Lookup(word);
    if (definition == NULL) {
        printf("  %s: ", word);
        definition = GetLine();
```

The window's left margin contains a stop sign where the user has placed a breakpoint with a simple click of the mouse. The line about to be executed has an arrow in the margin. Each of these icons is self-explanatory. Moreover, because the programmer is not confronted with a baffling array of more complicated options intended for advanced use, we find that students are much more willing to use the debugger than they have been in the past.

The debugger also includes a program visualizer modeled after the Macintosh Finder interface, which is familiar to Stanford students. The following diagram illustrates the operation of the visualizer in Thetis:

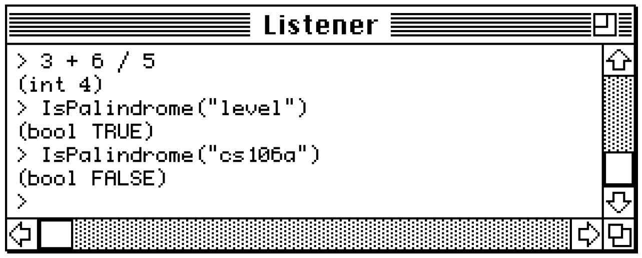


Each function on the call stack has a folding arrow next to it which can be turned down to view the variables in that function. Double-clicking on a function opens a new window with that function's variables in it. This metaphor extends to arrays and structures as well. The icons to the left of variable names represent the types of the variables: 1 for integers, abc for strings, and so on.

Our experience indicates not only that students find the visualizer easy to use, but also that its structure helps them develop a solid conceptual model of how their programs work. The scope of variables, parameter usage, and the concept of a call stack are all presented in an accessible, understandable way.

The Listener

Another debugging tool available to the programmer is the Listener, which allows students to evaluate any expression while the program runs. These expressions can include variables from the program, and even function calls, as illustrated in the following diagram:



```
> 3 + 6 / 5
(int 4)
> IsPalindrome("level")
(bool TRUE)
> IsPalindrome("cs106a")
(bool FALSE)
>
```

In this example, the **`IsPalindrome`** function is tested on several different input values. By using the Listener, students can test functions and determine the source of errors without repeatedly restarting the program or writing special code for running test cases. Providing the Listener also reinforces the notion of stepwise refinement by allowing students to verify that a newly written function works correctly before continuing to write additional code.

5. EVALUATION OF THE THETIS PROJECT

The Thetis interpreter was written by a small group of Stanford undergraduates (Stephen Freund, Andy Montgomery, Brian Becker, and Perry Arnold) over the past two years. The initial phase consisted of two components developed in parallel during the fall and winter of 1993–94: a preliminary parser/evaluator and a prototype debugger/visualizer. These components were integrated during the spring and summer to create the initial version of the interpreter. The fall of 1994–95 was dedicated to testing and refinement, using the participants in the seminar on teaching and the staff of the introductory course as the user community.

In winter quarter of 1994–95, we used Thetis on an experimental basis with a group of approximately 30 students in Stanford's large CS106A course, which covers the material in the ACM CS1 curriculum [Austing78]. This experiment gave us an opportunity to refine the system and fix any potentially damaging bugs before it faced widespread use.

That first quarter went extremely well. Through an informal survey, we found that most students enjoyed using Thetis and felt that the most important advantage was the run-time error checking on arrays and pointers, which saved considerable debugging time. Given the success of the initial experiment, we decided to use Thetis for the entire CS106A course (300 students) the following quarter. In addition, we allowed students in CS106B—Stanford's equivalent of CS2—to use either Thetis or THINK C for their assignments.

To measure student reaction to the tools, we conducted a survey of students enrolled in CS106A in the summer quarter of 1995, when, unfortunately, enrollments were relatively small. The results are as follows:

N = 39	Yes	No
In general, do you find Thetis easy to use?	100.0%	0.0%
Were you ever confused about how to do something in Thetis?	33.3%	66.7%
Do you feel that Thetis has helped you understand programming?	71.8%	28.2%
Do the error messages reported by Thetis help you find mistakes in your programs easily?	82.0%	18.0%

From the reactions, it is clear that student response to Thetis is positive. This same survey was also given to students enrolled in CS106X—Stanford's accelerated introductory course that combines the material from CS1 and CS2 into a single quarter—who were programming in THINK C. With this audience, the corresponding questions revealed the following:

N = 26	Yes	No
In general, do you find THINK C easy to use?	73.1%	26.9%
Were you ever confused about how to do something in THINK C?	69.2%	30.8%
Do you feel that THINK C has helped you understand programming?	73.1%	26.9%
Do the error messages reported by THINK C help you find mistakes in your programs easily?	19.2%	80.8%

Despite the fact that students in CS106X typically have a much stronger background in computing than their counterparts in CS106A, they reported having more trouble using THINK C than those using Thetis in CS106A. The most significant contrast concerns the usefulness of error messages. Over 80 percent of Thetis users said the error messages helped them find mistakes, while fewer than 20 percent said the same of THINK C.

When CS106B students were given the choice between environments, many students opted to use Thetis for their programs despite its slower execution speed. Often, students who started an assignment with Thetis would switch to THINK C to take advantage of the faster run times when they were doing the final fine-tuning of their programs. This practice constituted a valuable lesson in software engineering. Students learned how to program with portability in mind and also how to choose tools to fit the immediate task at hand.

Another indication that Thetis has successfully reduced the stress and frustration of students taking introductory programming classes are the questions asked by students at the helper cubicle in the main computing cluster. Students using Thetis were generally able to find common mistakes—such as not initializing a variable or accessing an out-of-bounds array element—on their own and therefore asked fewer questions. Thetis allows students to step over these small, yet often confusing and painful hurdles and concentrate on the main goal of learning how to program.

Mehran Sahami, a lecturer at Stanford, evaluates class performance on assignments by conducting a "pain poll." When students hand in an assignment, he asks the class collectively how painful it was and how much time they spent on it. When teaching with Thetis during the summer quarter of 1995, he noted that people generally thought the programs were less painful than usual. Perhaps more significantly, the amount of time required to finish each assignment dropped by one or two hours on the average. The assignments were no easier than usual, but the better support from the programming environment allowed people to move beyond simple misunderstandings and bugs more quickly.

The value of Thetis to the course staff is also apparent. Thetis allows section leaders to debug and grade programs more easily. It is also an excellent tool for presenting new topics during office hours or discussion sections. Lecturers use Thetis for demonstrations during class, supplementing lecture notes with real programming examples. Although we have no direct evidence to support the claim, our collective impression is that new ideas sink in better when lectures are augmented by demonstrations in Thetis.

6. CONCLUSIONS AND FUTURE PLANS

On the basis of our experience in the last year, we believe that Thetis fulfills the goals we set out to accomplish and that it provides a significantly better learning environment for students in CS1/CS2.

We are, however, continuing our Thetis development work and expect to improve the system in several important ways. Our primary technical challenges have been to increase execution efficiency and reduce memory utilization. The next version of Thetis, which will be released at Stanford in the fall of 1995, runs approximately five times faster and requires less than half the memory needed by the implementation discussed in this paper. We believe these improvements will allow students to take full advantage of Thetis without becoming frustrated by its slower performance.

We also intend to make the following enhancements to Thetis over the next two years:

- Extend the text editor to provide automatic formatting of the sort found in structure-based editors [Goldenstein89]
- Develop an effective on-line help system
- Provide more syntactic, stylistic, and run-time checking
- Enhance the visualizer to support in-class demonstrations
- Port Thetis to other platforms
- Release the system for general use

7. REFERENCES

[Austing79] Richard Austing, Bruce Barnes, Della Bonnette, Gerald Engel, and Gordon Stokes, "Curriculum '78: Recommendations for the undergraduate program in computer science," *Communications of the ACM,* March 1979.

[Birch95] Michael R. Birch, et al., "DYNALAB: A dynamic computer science laboratory infrastructure featuring program animation," *SIGCSE Bulletin,* March, 1995.

[CenterLine95] CenterLine Corporation, *CodeCenter User's Guide,* Cambridge, MA, 1995.

[Goldenstein89] D. R. Goldenstein, "The impact of structure editing on introductory computer science education: The results so far," *SIGCSE Bulletin,* September 1989.

[Hamilton42] Edith Hamilton, *Mythology,* Boston: Little, Brown and Co., 1942.

[Roberts93] Eric S. Roberts, "Using C in CS1: Evaluating the Stanford experience," *SIGCSE Bulletin,* March 1993.

[Roberts95a] Eric S. Roberts, *The Art and Science of C: A Library-Based Approach,* Reading, MA: Addison-Wesley, 1995.

[Roberts95b] Eric S. Roberts, John Lilly, and Bryan Rollins, "Using undergraduates as teaching assistants in introductory programming courses: An update on the Stanford experience," *SIGCSE Bulletin,* March 1995.

[Ruckert93] Martin Ruckert and Richard Halpern, "Educational C," *SIGCSE Bulletin,* March 1993.

[Schorsch95] Tom Schorsch, "CAP: An automated self-assessment tool to check Pascal programs for syntax, logic, and style errors," *SIGCSE Bulletin,* March 1995.