The new classical tiny-C

Thomas A. Gibson

January 7, 2018

Introduction

This writing serves two purposes. The body documents the product focusing on both usage and internals. It assumes knowledge of C, so on that subject it is terse, documenting the subset and differences. The internals includes a debugging technique, the test system, and machine calls. I am inviting you to both use and extend this work.

The appendix has related writings, or links to them: motivation, source documents, stories, and more. This is a bit of history, but tiny-C Stories, for example, explains why I varied from a strict subset of C, and other decisions.

The word CURRENTLY in all caps is a flag for me (or you if you would like) to someday do the modest enhancement implied.

The Language

Man page

NAME

tiny-C – classic 1977 tiny-C interpreter reimplemented in C, and GPL'd

SYNOPSIS

tinyc [ -d ] [LIBRARY] FILE

DESCRIPTION

Execute the tiny-C program in FILE using LIBRARY. Default library is pps/library.tc.

OPTIONS

-d debug mode

Syntax

Tiny-C's syntax is a lazy adaptation of the C syntax. The motivation for the variance is described in Stories, #3. The differences…

1. Comments begin with /\* and end at the end of the line. CURRENTLY // is planned but not implemented.
2. Function and variable names are internally canonicalized to the first 7 plus the last character. They are case sensitive.
3. Reserved names: if, else, while, return, break, char, int, MC, endlibrary
4. Data: int, char, one dimensional array of either. An array of dimension N has size N+1. The array syntax uses ()'s instead of []'s. So char player(9) has 10 elements, from player(0) to player(9). An int is signed 32 bit. Array names can be used as pointers. So player+3 will point to player[3]. And they are lvalue's, e.g. player=something.
5. Locals and Globals. All functions and all data not declared within a function are global. One required global is the application function main(). This differs from the 1977 tiny-C.
6. Library variables are all global variables before the special global word endlibrary. Look at the end of the supplied library for this use. A symbol (data or function) is looked up in this order: locals, globals, library.
7. Operators: + - \* / % < > <= >= == != =
8. Expressions: The top level expression parse is assignment. Then relation, etc, down to factor and constant, the same as C. So while( k=(k+1)<MAXK ) [..code..] works just fine.
9. Functions: Function bodies use [] instead of { }. Parenthesis are optional for arguments provided the function is the last of an expression. So foobar data, another is the same as foobar(data,another). But that comma is required even in the first form. And number=gn is fine, but number=(gn()+10)/2 requires the ().
10. Newlines and semicolon separates statements. So hello; goodbye; seeYa tomorrow calls hello, calls goodbye, calls seeYa(tomorrow).
11. Statements: if, else, while, return, break. If, else and while bodies use []s instead of {}s.
12. No semicolon required for the last statement on a line, as the end-of-line (or beginning of a comment) also ends the statement. Semicolons also separate statements, so multiple statements can be on one line.

Library

The standard library is in the file pps/library.tc, and is written in tiny-C. CURRENTLY the function definitions are in comments preceding each function, but will be here someday.

Debugger

Normal Usage, examining behavior of tinyC code

To use the debugger, start tiny-C with the -d option:

$ tinyC -d <filename>

You will get a debug prompt

(db-tc)

The commands are single letters, space (optional), single argument (for some commands)

|  |  |
| --- | --- |
| b <symbol> | set a breakpoint at the in-scope symbol. CURRENTLY: the symbol must be in canonical form, first seven letters plus the last. |
| i [b] | information: CURRENTLY only a list of breakpoints and their state |
| r | run the program from the beginning. CURRENTLY same as c (continue) except for its first use to start the run. |
| p <variable> | print the value of the in-scope variable |
| n | next statement. |
| c | continue running to next breakpoint. |
| x | exit the interpreter, printing the final status. |
| g | go to debugger under which the interpreter is running, e.g. to (gdb). See the section documenting this feature. Used to find interpreter bugs. |

Usage notes

This tiny debugger is motivated by my experience with gnu's C/C++ debug, gdb. Seven commands mimic those of gdb, but there are differences.

Breakpoints are set on symbols that must be in scope. That usually means globals, but locals provided their declaration has been processed. All function names are global, so they can always be breakpointed. Any occurrence of that symbol causes the debugger to be entered before the symbol is processed. The prompt (db-tc) is displayed and you can enter any of the above commands. Use the c command to continue running. CURRENTLY the r command also continues running but it should offer a query to restart.

Information with or without the b argument prints a table of current breakpoints. Note that a breakpoint on a local that goes out of scope will disappear from the table. It will not reappear if the function is re-entered, and if needed must be reset. CURRENTLY enabled is always 'y' (yes). CURRENTLY hits are not counted. CURRENTLY the v option (print names of in-scope variables) is unavailable.

Print a variable is probably the most useful command. Its value is displayed. CURRENTLY arrays are fully printed, so it is best to keep their dimensions small until debugging is complete.

Run starts or continues the already loaded tiny-C program. No argument is needed. CURRENTLY it does not offer the restart option, so a second use of r does a continue.

Next does the next statement. This differs from gdb which does the next line with an executable. The interpreter is statement oriented, not line oriented.

Continue continues execution until another break occurs, or the program ends. If the program ends on an error it is printed, and the interpreter exits.

Exit terminates interpreter execution.

Examining interpreter code

The debugger is part of this interpreter implementation. So you can examine the debugger/interpreter by running it all under your favorite debugger. It need not be gdb, although that is what I used. So under your favorite debugger for C code, follows the steps outlined below.

The issue is shifting back and forth from one debugger to the other. Here is how I did that...

Using tc debug UNDER your favorite C debugger

setup

1. In the tinyC code whatever.tc declare a dummy variable, here called foo, and put a dummy statement, foo=1 will do, in some function where you want to examine behavior.
2. From within your favorite debugger start tinyc -d with your subject code, tinyc -d whatever.tc. Note that startups under different debuggers differ. Mine, gdb, can only load code at startup, and arguments are given later. But the intention is to get some breakpoints set, then get it all ready to run, but not yet running with the -c and the whatever.tc arguments.
3. TinyC has a function gdb\_b which does nothing but is invoked by the g command in debug. Set a breakpoint on this function, b gdb\_b. This enables getting out of tinyC's debugger back to your C debugger.
4. Now start it all running, run -d pf.tc, or your equivalent. The -d argument means you will see the tinyC debugger prompt, (tc-db).
5. Now set a breakpoint on the step 1 dummy variable, foo, (tc-db) b foo. All is arranged.

session...

1. Use the r command to start the interpreter running. (tc-db) r. The interpreter takes over, and your tc program runs until it hits the reference to foo wherever it was planted. At this point you are navigating the whatever.tc code, examining variables, setting more breakpoints, etc.
2. When you get to a point where you want to examine interpreter code the g command invokes the do-nothing function g\_db. Now you are in your C debugger examining the interpreter code. One, or perhaps a few (5 for gdb), single steps will get you out of g\_db into interpreter code.

The Test System

A second executable, test, and its source code is part of this project. It includes a battery of interpreter and library tests. Most can be run in batch mode for quick regression testing. A few others involving interaction must be run manually and their outputs compared with documented good results.

Man page

NAME

test – A program that runs tests on tc.c focusing on internals.

SYNOPSIS

test [TESTNUMBER ...]

DESCRIPTION

With no arguments test runs all tests that require no interaction, those with numbers less than 90. With numbers it runs just those tests. The simpler tests have numbers in the 10's and 20's. Interaction numbers are in the 90s.

MAKE COMMANDS

These commands facilitate frequent tasks...

make dotest – Run all regression tests producing a test\_results file.

make diff – diff the test\_results and good\_results files, results to stdout.

make keep – copy test\_results to good\_results

Usage Notes

As of this writing there are 46 tests organized for simple regression testing. The are numbered 1 up but less than 90. Many are coded completely in the file test.c, but a few use files stored in the directory testFiles.

To run tests

./test [TESTNUMBER ...]

With test numbers just those tests are run. If none are given ALL tests are run. The test results are printed to stdout. This is useful for just one or a few tests.

A more convenient way to run all tests is...

make dotest

which produces a file, test\_results. Follow that with...

make diff

which does a unix diff on test\_results against good\_results to stdout. I do not redirect that but let it print to my terminal. Then scroll up/down to find issues. Every test prints a header like...

TEST 16 09:11

which includes a time stamp. So the different time stamps for good\_results vs test\_results diff will always cause both headers to print...

< TEST 16 09:11

> TEST 16 11:15

The times themselves are irrelevant. Actual result differences ARE relevant, and this simple trick identifies which test needs attention. After doing a make diff scroll up and down and if all is cool it takes 5 seconds to determine that.

The good\_results file is the real oracle for the regressions. There used to be lots of pointers in the results which would drift as code was changed, causing false differences. I removed most of them, but some pr[..]'s still have differences, e.g.

164c164

< stack entry at 2: 0 A 1 pr[99]

> stack entry at 2: 0 A 1 pr[67]

In test.c every test has a comment labeled "Should get…" which also shows the expected results. For the non-regression tests, in the 90's range, these comments are the oracle. Run a specific test, do the interaction, and by eye compare the test results with the Should get comment. Fortunately there are presently only four such tests.

New Tests

Their are three basic coding styles for tests.

The early tests are hard coded as cases in a giant switch statement, test.c lines 120..750. A testSetup function is available for initialization. Tests 1 through 29 and a few 30's use this style.

The second style reads a single statement from a file and runs just that statement. Test 30 is an example. The single statement can be compound, so in fact a substantial code fragment can be done this way. The function testSetupFile does the initialization including an optional library.tc load, and the file read.

The third style is a variation on the second, but the file is a whole program, with main[] and functions. The test case code uses testSetupFile but also calls link(). Test 43 is an example. But later tests use testWhole which calls link to initialize. That call to link whether hard coded in the case code or from testWhole distiguishes the third style from the second.

Machine Calls

Machine calls are coded in C but can be called directly from a tinyC program. Best practice is to wrap the MC in a library or private routine. They are numbered in three ranges: originals (1977), new (2017), user defined. Add your personal MCs to the third range, numbered 201 up.

The arguments are parsed and stacked, with the last argument being the MC number. For example…

/\* Put a character to the terminal

putchar char c [

if(c==0)c='"'

MC c,1

]

The implementation is…

McList origList[] =

{ &Mpc, &Mgch, &bar, &bar, &bar

...

};

and

int Mpc(int nargs, int \*args)

{

printf("%c", \*args);

}

MC 1 acquires its number by its placement in the origList array of function addresses. Its implementation routine is named there. The MC infrastructure pops all the args into nargs, \*args. And it pushes whatever your implementation returns as an int.

Follow this model, and build yourself a nice library.

Sample code

Find sample tc programs in...

<https://github.com/tgibson37/tiny-c/tree/master/SamplePrograms>

Currently just piranha fish (pf.tc).

Appendix

A. tiny-C stories

Find these in <https://tinyurl.com/tinycstories>.

B. Implementation Notes

Porting Considerations

The 8080 is a byte oriented typeless machine. It does have some 16 bit instructions treating a pair of 8 bit registers as a 16 bit register, and that is as close as it gets to having types. A 16 bit datum may be an integer or it may be an address, a pointer. Neither the hardware nor the assembler knows the difference or cares. It is how they get used that mattered. This was reflected in the 8080 interpreter code.

C is typed. Reading the 8080 code and reverse engineering into C code meant adding this typing information into the code. But lots of the 8080 code simply did not use the datum, just passed it along. The 8080 code carries type information alongside each datum or array in the entries to the stack and the variable table. Alongside are type, class, and len which describe the type. Data is separated from this only when it is going into code that implicitly knows the type. A good example is toptoi(), tc.c line 130. It returns an int which may in fact be a tiny-c char, int, or array reference (pointer). In C, char and int can be assigned to each other without casting. And for 32 bit compilers, so can pointers and ints. But when the datum ultimately gets used, its tiny-c type must be known. Toptoi is typeless, but can only be used where the type expected is known.

Entries in both the calculation stack, stack, and the value storage area, pr (beyond the tc code), are typeless values, just bytes. My implementation uses a union of int, char, and void\* into one typeless container, union stuff. See tc.h lines 40..68. Both stackEntry and the variableEntry have union stuff to hold values. Datum are moved between stuff and C variables obey type, but once in stuff they can be passed around in a typeless manner. That technique made it easier for me to translate 8080 code into C.

Awareness of this design issue is probably the most important fact not answered by inspection of the code.

One anomoly: For some reason my compiler compiled 64 bit pointers. The high 32 bits never get used, just space for them compiled in. Toptoi returns a 32 bit int, which is often cast to a char\* or int\*. It works. Why my compiler allocates those 32 usless bits I do not know. Be warned this may affect a porting effort.

My environment

I built this version of tiny-C on mint linux using these tools:

- gnu's c compiler, gcc ubuntu 4.8.4-2

- gnu's make 3.81, makefile

The makefile has all the technical specs. For example this line...

CFLAGS = -w -g -ansi -I /usr/lib/syslinux/com32/include/

defines all the dash flags used on most compile/link.

One anomaly. Two files would not compile with those flags, I don't know why...

- time.c

- FileRead.c

But they compile and work fine with NO flags except -c, just pure gcc -c time.c, for example. And the link phase found their objects just fine. The makefile handles this anomaly.

Make install just copies the executable and library.tc files to where I want them on my linux box. Change this to suit your environment.

I am not familiar with either Windows, or Apple's OSx. I assume they have build facilities not too far removed from my supplied make. So if you are porting my work to another environment you may have a bit of work to port the makefile, especially the compile flags.

Archive

Status

This is STATUS.txt in <https://github.com/tgibson37/tiny-c>.

License to this work

This is LICENSE.txt in <https://github.com/tgibson37/tiny-c>.

8080 code

This is tc.asm in <https://github.com/tgibson37/tiny-c>.

My Development Environment

This is MY\_ENVIRONMENT.txt <https://github.com/tgibson37/tiny-c>.

Background

This work started out as a pure nostalgia project. In 1977 I designed and coded an 8080 software product, tiny-C, an interpreter for a subset of the C computer programming language. To this day I look back on tiny-C as one of my most fun technical accomplishments.

From time to time I indulge in looking back to my more youthful projects, and one day browsing on-line what a pleasant surprise: I discovered the facebook site “Not just tiny-C,” hosted by Lee Bradley. I joined, of course, and contributed eight very short “tiny-C stories” plus a few comments from time to time. The stories are in this writing, Appendix A.

My browsing included an 8080 CPM I had written in 1977-78, and reading a bit of it was fun. Forty year old assembler code, and not a few mysteries. Soon I was taking notes. Soon the notes looked like C pseudo-code. And soon I was typing it in, and even compiling small segments. I was getting sucked into a larger project. Soon I was working towards a full implementation of tiny-C in C.

Another motivation was a question addressed to me on Not Just tiny-C, did a licensable version of tiny-C exist? The answer was basically “no”. I had formed a partnership, Tiny-C Associates, and my partner, Scott Guthery and I had developed the marketable version, all copyrighted by the partnership. That Association no longer exists, and its properties are all “orphans”. But Scott and I agreed each could use the properties independently. He and I both wrote books about tiny-C, his being by far the better. But a cleanly licensable version needs more than that agreement. The easiest solution would be for us to reconnect and agree to GPL all the associations properties. But my attempts to locate Scott have not yet succeeded.

But one property, my 8080 source code, is copyrighted by myself, and the copyright is right in the source code. That was my starting point for my full implementation.

Links

Where to find this project: <https://github.com/tgibson37/tiny-c>

Lee Bradley's “Not just tiny C”: <https://www.facebook.com/groups/299317782048/>