Interpreted C
as a
Scripting Language

EXTENDED ABSTRACT

John David Duncan,
Oregon Public Networking
jdd@efn.org

Abstract

This paper examines **cint**, an interpreter for C and C++ developed by Masaharu Goto at HP Japan, and some of its potential applications in systems administration. It explores the rationale behind developing a C and C++ interpreter, and the potential advantages of creating administrative tools in interpreted C. I describe my experiences obtaining cint source code, installing it on various platforms, and developing several small applications. Cint is illustrated using some scripts which focus on its use as an interface to system calls and to pre-compiled object libraries, including a simple interface to *.dbm* files. Finally, an attempt is made to evaluate the performance of cint in comparison with compiled C, Java, Perl, and Awk.

Cint

Cint is a C interpreter which is written strictly in ANSI C and which implements a subset of ANSI C. It provides a C++ interpreter, as well, when it is built on platforms that provide a C++ compiler. Cint has been succesfully built on many operating systems, including unix, Microsoft Windows, MacOS, MS-DOS, and BeOS. It is capable both of interpreting C source code of compiling loops in interpreted code into bytecode for faster execution. Cint can invoke a system's standard C preprocessor or C compiler for on-the-fly compilation of all or part of a source file. It can link interpreted scripts against pre-compiled dynamically-loaded object files, and it can execute pre-compiled object code using symbols defined inside an interpreted source file . The cint package includes an interactive debugger. When built with the GNU Readline library, cint itself can be run interactively as an extensible C++ command shell environment which allows dynamic loading and unloading of C++ objects and provides run-time type information. This paper examines only a few of these features.

Cint claims to be "95% compliant with ANSI C and 85% compliant with ANSI C++." It lacks support for a few operators (e.g. the comma operator in C) and standard library functions (setjmp(), longjmp(), and the family of va_arg() and vprintf() routines). Nonetheless it is reportedly robust enough to interpret its own 60,000-line source code. It includes a few features aimed at making C more convenient as a scripting language, such as automatically initialized variables. While the distributed cint is a bare-bones ANSI C interpreter, it is readily extensible. Cint itself, along with a tool called makecint, can be used to create a custom cint executable that supports functions from arbitrary object files, or to create shared object files that may be dynamically linked with the standard cint at runtime. (People familiar with TeX might find this process reminiscent of using *initex* to create a custom version of *tex* such as *latex*).

The main cint developer is Masaharu Goto (gotom@jpn.hp.com). The source code can be obtained from *http://hpsalo.cern.ch.root/Cint.html* ; the current version (in May, 1999) is 5.14. It is distributed under a licensing agreement which reads "Commercial use of CINT requires registration to Hewlett-Packard Japan. Send e-mail to the author (gotom@jpn.hp.com) describing your name, company or organization, address and purpose of using CINT."

ROOT

Unix system development fundamentally involves creating executable objects – ".o" and ".so" files – and linking them together. In such "low-level" development, these files are often created from C source code. However, this level of systems development is seemingly separated from a higher level that is usually implemented in the shell and in scripting languages such as awk and perl. In order to build a bridge between these two realms, systems administrators may sometimes write trivial wrappers around library routines, such as the *printf(1)* shell command. Shell commands are strung together in scripts, with some overhead cost required in spawning a separate process for each task. When the performance of high-level systems becomes insufficient, they must be largely rewritten in C in order to be implemented as speedier low-level systems. In exploring cint, my initial goal was to see whether it might bring the two realms closer together, so that low-level and high-level development can occur in a common language and so that the interface between them might be simpler.

Cint is currently deployed as a fundamental component of the ROOT project [Brun 1997], a "comprehensive object-oriented framework on which large scale data analysis applications can be built," based on C++ and being developed at CERN. The ROOT framework provides statistical analysis and graphing functions for large data sets and is based on a customized version of cint called *rootcint*. As C++ objects are developed and accepted into the framework, *makecint* is used to create a new version of *rootcint* that includes them. Applications built on top of ROOT in C++ may either be interpreted or compiled. As the authors of ROOT note:

> "CINT as embedded in ROOT can be used as command line interpreter and as macro processor, where macros are 'small' (up to at least 60000 loc) C++ programs. Thanks to CINT the ROOT system can present the user a single language environment: C++ as implementation, macro and command line language.
>
> "The advantages of a single language model are clear. Especially when writing macros. Typically macros start as small prototypes that need frequent modification. While execution speed is not important a short edit-execute cycle is. However, once macros have grown to full programs and have become stable, the need for fast execution in production jobs becomes important. Thanks to the fact that the macros are in 'standard' C++ we can simply compile and dynamically link the macros with the ROOT system and execute them at full speed." [Rademakers 1997]

Many of the goals that might be achieved through cint can also be accomplished with Perl. Perl is certainly a more mature interpreter, with many more users. The mere fact that perl provides an easy interface to a large number of system calls accounts for a great measure of its utility. Perl 5 is extensible via easily-created modules, a vast number of which are available from CPAN. However, there is no reason that the availability of Perl should discourage research and development of Interpreted C and C++.

Installing Cint

I obtained Cint from the ROOT www site and installed it on three platforms -- SunOS 4.1.4, BSD/OS 4.0, and Linux 2.0. The *cint.tar.gz* file should be unpacked into the eventual root directory of the cint system; on unix-variants, /usr/local/cint/ is the standard choice. This will become the CINTSYSDIR.

Under cint's platform/ directory live the description files used to build cint on various platforms. Porting cint to a new architecture is accomplished by creating an appropriate description file as per the documentation in platform/README.txt. Cint is built using a script called "setup", which takes the name of a platform description as an argument. Cint 5.13 turned out to have some problems on my BSD/OS test platform. I sent bug reports to the author and received patches back within a day; all that is required to build *cint* 5.14 and *makecint* under BSD/OS is the command "sh setup platform/bsdos". Non-unix installation may be less straightforward. Building cint under Windows requires Visual C++, Symantec C++, or Borland C++. Building it for DOS requires djgpp. As for BeOS and Mac builds the platform/README.txt file directs the reader to consult Fons Rademkaers at CERN. Once cint is built, under unix, you are instructed to set the following environment variables.

```
CINTSYSDIR=/usr/local/cint
PATH=$PATH:$CINTSYSDIR
```

In order for the dynamic loader to locate precompiled cint libraries, it may also be necessary to set:

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$CINTSYSDIR
```

Platforms that use LD_ELF_LIBRARY_PATH will need to set that as well.

Under CINTSYSDIR, header files live in include/, there is some documentation under doc/, and a large selection of demonstrations and examples are under demo/. Once cint is installed, it is capable of running a simple script like this, the equivalent of the GNU command *date --date "6 hours ago"*:

```
bash% cint -x '
 #include <time.h>
 main() {
 int t;
 time(&t);
 t -= 6*60*60;
 printf("Six hours ago was: %s\n",ctime(&t));
 }'
```

However, the freshly-installed cint is still not capable of performing a great number of common tasks. For instance, it does not support the standard library of Berkeley socket calls. Fortunately, within the cint's lib/ directory are a few examples of extensions that can be added to cint, including a socket library. To expand cint's capabilities, enter the CINTSYSDIR/lib/socket/ directory and run the "setup" script there. Setup invokes *makecint*, which, in essence, is a simple program merely capable of parsing its command line arguments and creating a Makefile. When the setup script runs the command:

```
makecint -mk Makefile -dl $CINTSYSDIR/include/cintsock.dll -h cintsock.h -C cintsock.c
```

the output of makecint looks like this:

```
#####################################################################
# makecint : interpreter-compiler for cint (UNIX version)
#
# Copyright(c) 1991~1996 Hewlett-Packard Japan
# Author                 Masaharu Goto (gotom@jpn.hp.com)
# Copyright(c) 1995~1998 Masaharu Goto (MXJ02154@niftyserve.or.jp)
#####################################################################
Makefile is created. Makecint success.
Do 'make -f Makefile' to compile the object
```

The newly-created Makefile can be used to build a dynamically-linked object file called *cintsock.dll. ciintsock.dll*, in turn, will allow cint to implement the functions in *cintsock.c* via the interfaces defined in *cintsock.h.*

Upon running "make" to extend cint, the output (under SunOS) looks like this:

(1) `gcc -DG__SUNOS4 -O -fpic -o cintsock.o -c cintsock.c`
(2) `cint -K -w1 -zcintsock -nG__c_cintsock.c -D__MAKECINT__ -DG__MAKECINT -c-2 -DG__SUNOS4 cintsock.h`
(3) `gcc -I/usr/local/cint -DG__SUNOS4 -O -fpic -c G__c_cintsock.c`
(4) `g++ -shared -O -DG__SUNOS4 -o /usr/local/cint/include/cintsock.dll cintsock.o G__c_cintsock.o`

First, the socket routine code is compiled to object code. Then, the magic "cint -c-2 ..." incantation on on line *(2)* tells cint to create a file *G__c_cintsock.c* containing the source code for cint's interface to the functions whose prototypes appear in cintsock.h. In line *(3)* G__c_cintsock.c is compiled to object form. Where the socket libraries themselves contain symbols like *_accept* and *_bind*, G__c_cintsock.o contains the analagous symbols *_G___accept_7_0* and *_G__bind_8_0* used internally by the interpreter to execute references to the calls.

Once the whole gamut makecint –> gcc –> cint –> gcc has been run, *socket.h, cintsock.h,* and *cintsock.dll* all exist under $CINTSYSDIR/include. It is now possible to use the socket library in a script, like this:

```
#!/usr/local/cint/cint
#include <socket.h>
main() {
   ...
}
```

Cint's <socket.h> file contains a surreptious *#pragma include "cintsock.dll"* directive, instructing the interpreter to link itself against our newly-created socket library.

Cint Peculiarities

(Outline of this section:

1. *automatic initialization*
2. *#pragmas*
3. *Portability: Code should be portable from cint on platform A to cint on platform B, but not necessarily portable from platform A's native C compiler to Cint on platform A. However, the use of makecint to create customized environments might disrupt cint-to-cint portability across platforms. It should be possible to define a standard cross-platform interface layer – for instance, a layer that provides socket services and hides the fact that they may be implemented either as Windows sockets or as unix sockets. This is only natural in C++, and is not out-of-reach in C.*

)

Some Sample Scripts

(Note to eds.: the rest of this paper is still in very rough form)

Ex.1: Using the *gdbm* library.

In this example, makecint is used to incorporate support for GNU gdbm 1.8.0 into cint. First, build gdbm; then, from within gdbm's source directory, run makecint, like this:

```
makecint -mk cintmkfile -p -dl $CINTSYSDIR/include/gdbm.dl -h gdbm.h -l libgdbm.a
make -f cintmkfile
```

This invocation of makecint includes the *-p* flag, which tells it to invoke the standard C preprocessor rather than its own somewhat crippled one, and *-l libgdbm.a* to link in the already-compiled GDBM library. Once gdbm.dl is built, we can then execute the following code. Note that the apparently-quoted shell parameters ($1, $2) are filled in before the code is interpreted.

```
#!/bin/sh
# fetch [key] [dbfile]  -- fetch a value from a gdbm
cint -x -p '
#include <stdio.h>
#include <gdbm.h>
#pragma include <gdbm.dl>
main() {

GDBM_FILE dbf;
datum key, value;

if ((dbf=gdbm_open("'$2'",0,GDBM_READER,0,NULL))==NULL) {
```

```
        puts("Cannot open gdbm file '$2'")
                        exit(1);
        }
key.dptr="'$1'";
key.dsize=strlen("'$1'")+1;
value=gdbm_fetch(dbf,key);
gdbm_close(dbf);
if(value.dptr==NULL) exit(1);
printf("%s\n",value.dptr);
exit(0);
}'
```

Ex.2: Adding support for a new system call.

This example was also inspired by a small database application – this time, using flat textfiles for data, rather than dbm files, and maintained entirely via shell scripts. While our data files received only a few transactions per hour in the initial implementation of this system, we nonetheless encountered a need for file-locking. We also wanted to provide each data transaction with a unique tracking number. These tasks were both accomplished using a simple shell command called *locktime. locktime* attempts to create a lockfile, and, if it succeeds, it also returns the system time (as an integer) on standard output. The time is used as a tracking number, and it is guaranteed to be unique so long as any process which obtains a lockfile promises to hold it for at least one second before releasing the lock. Thus, scripts that use *locktime* look like:

```
TRACKING_NO=‘locktime db.lock || exit‘
        [ ... perform database transaction ... ]
sleep 1
rm -f db.lock
```

This was the initial version of *locktime*, which relies on `open(O_CREAT | O_EXCL)` in order to guarantee an exclusive lockfile:

```
/* locktime.c -- create a lockfile and print the system time */
/* Usage: locktime _LOCKFILENAME_ */

/* open the file O_CREAT | O_EXCL, then get the system time & return it */

#include <fcntl.h>
#include <time.h>
#include <errno.h>
#include <stdio.h>
#define NAPTIME 1
#define MAXTRIES 6

main(int argc, char *argv[]) {
time_t date;
int tries=0, fd;
extern int errno;

if (argc != 2) {fprintf(stderr,"usage: %s LOCKFILENAME\n",argv[0]);
                exit(1);
}

while((fd=open(argv[1], O_WRONLY | O_CREAT | O_EXCL, 0666)) == -1 &&
        errno==EEXIST) {
                if (++tries >= MAXTRIES) break ;
                else sleep(NAPTIME);
        }

if ((fd==-1) || close(fd)==-1) {perror(argv[1]); exit(1); }

date=time(NULL);
printf("%d\n",date);
}
```

Since *locktime* is the only compiled code in this application (running on a heterogeneous network of both SunOS and BSD/OS servers), it is tempting to implement it in cint. Unfortunately, *open(2)* is a Posix system call rather than an ANSI C function; the stock cint doesn't support it. It is again necessary to use makecint, although in this case there are no actual object files to link against. *open()* is defined in fcntl.h (or actually, for SunOS, in fcntlcom.h); the header file alone is sufficient for cint to create an interface method. So Makecint is invoked as:

```
makecint -mk Makefile -p -dl fcntl.dl -h fcntlcom.h
```

After some tweaking of the stock (pre-ANSI) SunOS header file, running "make" creates a fcntl.dl file allowing cint to implement the fcntl routines. Here's the final interpreted version of *locktime*:

```
#!/bin/sh

CINTSYSDIR=${CINTSYSDIR:-/usr/local/cint}
export CINTSYSDIR
NAPTIME=1
MAXTRIES=6
LOCKFILE=$1

$CINTSYSDIR/cint -x -p -N fcntl.dl '
#include "fcntlcom.h"
#include <time.h>
#include <errno.h>
#include <stdio.h>

main() {
int fd;
extern int errno;

while((fd=open("'$LOCKFILE'", O_WRONLY | O_CREAT | O_EXCL, 0666)) == -1 &&
        errno==EEXIST) {
                if (++tries >= '$MAXTRIES') break ;
                else sleep('$NAPTIME');
        }

if ((fd==-1) || close(fd)==-1) { perror("'$LOCKFILE'"); exit(1); }

date=time(NULL);
printf("%d\n",date);
}'
```

The -p option in cint (as in makecint) invokes the standard preprocessor; *-N fcntl.dl* is equivalent to *#pragma include fcntl.dl.*

(This example is still under development – Masaharu Goto has recently suggested to me via email that, rather than using the fcntl.h files from my test platforms, I should add POSIX-based CINT-specific fcntl functions to CINTSYSDIR/lib/posix. If this effort is succesful, the support may actually be included in the cint distribution by the time this paper is presented).

Performance Comparisons

Kernighan [1997] and Van Wyk expound on some of the many difficulties encountered in attempting to compare the performance of scripting languages. They find "enormous variations" in sampled execution times, concluding that "there seems to be little hope of predicting performance other than in a most general way; if there is a single clear conclusion, it is that no benchmark result should ever be taken at face value." Nonetheless, they offer as a general guideline, that "compiled native code (C) runs fastest; next fastest are interpreted byte codes (Java, Limbo, Visual Basic); next come interpreters that construct and execute an internal representation like an abstract syntax tree (Awk, Perl); slowest of all are interpreters that repeatedly scan the original source (Scheme, TCL)." (p.6)

Cint contains an interesting mix of these characteristics. While code in cint is, in general, interpreted, most *for(), while(),* and *do()* loops are compiled into bytecode at runtime, and often some code is either compiled for execution or precompiled and

linked. It is plausible to hope that cint's performance will at least be comparable with that of Java, Perl, and Awk -- that is, in the same order of magnitude -- for some common tasks.

(At this point, I will reproduce the tests from Kernighan[1997] on my test platforms, from the original source and data, using compiled C, awk, perl, and interpreted C. This battery of programs is designed to test speed of basic language features like loop iteration and integer math, array and string performance, and I/O. Preliminary results of these tests, in comparison to Kernighan and Van Wyk's results, are quite odd -- on my machines, perl appears to have a much higher startup time than either awk or cint, and cint is faster than the other two over many iterations of a loop, but apparently has worse I/O performance.

Next, I take the Markov application from Ch. 3 of The Practice of Programming [Kernighan 1999] and time both the C and C++ versions in cint. Some alterations to the original source are required, as Kernighan & Pike's error-handling routines rely on va_arg(), which is not supported in cint, and their C++ code uses STL, which, while present in cint, is sketchy. The results are compared to compiled code, perl, and awk on my test platforms, and to the published performance times on p.81. I may also test the Java implementation, but so far I don't have a JVM available on any of my test platforms.

Finally, I'll do a few tests to compare runtime performance of static linking vs. dynamic linking of cint add-ons.)

Conclusions

Cint is still immature, and still under active development. Most of the problems I encountered installing cint 5.13 on my three platforms were fixed a few weeks later in cint 5.14.

My original goal was to find an environment where compiled *.o* files could be immediately utilized in the familiar, interpreted, shell command environment without further compilation. Cint does not obtain this goal. It is always necessary create the source code for an interface to the functions within an object file, to compile that interface support, and then to link cint to the newly compiled interface either statically or dynamically. However, this rather elaborate process has been highly automated, generally reduced to the two steps of running "makecint" followed by "make".

The two sorts of linking, static and dynamic, suggest two different models by which cint can be extended into a vital component of a computer system. In the static model, as developed in ROOT, a complete application framework is implemented as a customized version of cint. In the dynamic model, for each *.h* header file on a system, there is a corresponding shared object (*.so*, or, as presented here, *.dl)* file under CINTSYSDIR; this file contains cint's interface to the routines established in the header, and is loaded dynamically by cint when needed.

Cint is more useful the more tightly it is integrated with its host operating system; it would be most useful if OS vendors were to include both a generic cint and an OS-specific cint as part of the base OS distribution.

Acknowledgments

Many thanks go to Masaharu Goto, the creator of cint, who has always responded promptly to my bug reports and requests for help. Selena Brewington at the University of Oregon provided server space for a test platform, and lots of other help. Oregon Public Networking also provided test platforms. Jay Jones proofread a draft of this paper and offered many helpful suggestions.

References

[Brun 1997] Rene Brun and Fons Rademakers, “ROOT - An Object Oriented Data Analysis Framework,” Proceedings AIHENP ’96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86. See also http://root.cern.ch/.

[Kernighan 1997] Brian W. Kernighan and Christopher J. Van Wyk, “Timing Trials, or, the Trials of Timing: Experiments with Scripting and User Interface Languages” *http://www.cs.bell-labs.com/cm/cs/who/bwk/interps/paper.ps*

[Kernighan 1999] Brian W. Kernighan and Rob Pike, The Practice of Programming , Addison Wesley Longman

[Rademakers 1997] Fons Rademakers, “The CINT Interpreter Interface”, http://hpsalo.cern.ch/root/CintInterpreter.html