EiC

Contents

EiC is designed to be a production tool, it is not to be viewed as a toy, and is certainly one of the most complete, freely-available C interpreters built to-date. It is suitable as: an aid in teaching C, for fast prototyping of new programs and as a research tool — as it allows the user to quickly interface and experiment with user supplied, standard ISO C and POSIX.1 functions via immediate statements, which are statements that are executed immediately.

EiC can be run in several different modes: (1) interactively, (2) non-interactively (3) in scripting mode and (4) it can be embedded in other systems.

0.1 Interactive mode

In interactive mode, the user enters commands, or immediate commands, at the EiC prompt. Each immediate instruction produces a type, even if the type is void; as for example, C statements, declarations etc. All resulting type values are displayed:

```
EiC 1> 3*55.5;
        166.5
EiC 2> "hello, world!";
        hello, world!
EiC 3> int i;
        (void)
EiC 4> for(i=0;i<10;i++);
        (void)
EiC 5> i;
        10
EiC 6> struct {int a; double b[3];} ab = { 5,{0,1,2}};
        (void)
EiC 7> ab;
        {5,Array}
```

```
EiC 8> ab.a = 3;
        3
EiC 9> ab.b[2];
        2
EiC 10> #include <stdio.h>
        (void)
EiC 11> printf("hello\n");
hello
        6
```

0.2 EiC is pointer safe

EiC is also pointer safe. This means EiC catches most types of array bound violations; for example (for brevity, some output has been deleted):

```
EiC 1> int a[10], *p, i;
EiC 2> a[10];
READ: attempted beyond allowed access area

EiC 3> p = &a[5];
EiC 4> p[-5];
EiC 5> p[-6];
READ: attempted before allowed access area

EiC 6> p[4];
EiC 7> p[5];
READ: attempted beyond allowed access area

EiC 8> *(p+100);
READ: attempted beyond allowed access area

EiC 9> p = malloc(5*sizeof(int));
EiC 10> *(p+100);
READ: attempted beyond allowed access area

EiC 11> for(i=0;i<100;i++) *p++ = i;
WRITE: attempted beyond allowed access area
```

To detect array bound violations as efficiently as possible, EiC does not concern it self with the values held or produced by pointers, it only worries about address values when they are either referenced or dereferenced:

```
EiC 1> int a, *p;
EiC 2> p = &a;
EiC 3> p+10;    // okay, no problems
EiC 4> *(p+10);   // but just try to read or write to the address
READ: attempted beyond allowed access area
```

0.3 Running EiC non-interactively

EiC can also be run non-interactively or in batch mode, where it is possible to run C programs in a typical interpreter style. It can also handle programs that accept command line arguments, as seen from the toy example in main2.c:

```
#include <stdio.h>
int main(int argc, char **argv)
{
    while(argc--)
        printf("%s\n",*argv++);
    return 0;
}
```

The first parameter, argc, holds the number of argument strings passed to the program and is always at least one. The second parameter, argv, is an array of unspecified size of pointers to the input strings, which the first one will be the name of the program being executed:

```
% eic main2.c 123 hello -Dworld this.and.that
main2.c
123
hello
-Dworld
this.and.that
```

0.4 EiC's scripting language

In non-interactive mode, EiC runs generally like a typical interpreter, accepting input from a complete C program. However, EiC is also a scripting language. Below is an example of an EiC script, called `hello.eic`:

```
#!/usr/local/bin/eic -f
```

```
#include <stdio.h>
printf(" ******* Hello from EiC's script mode. ******\n");
```

The `-f` command-line switch, informs EiC to run in script mode. In script mode, EiC will treat all lines beginning with '`#`' and which cannot be interpreted as a preprocessor directive as a comment. To run the above script and assuming that it's executable (chmod +x hello.eic):

```
% hello.eic
 ******* Hello from EiC's script mode. ******
%
```

Another example of a more extensive EiC script is given in `script1.eic`:

```
 1    #!/usr/local/bin/eic -f
 2    #include <stdio.h>
 3
 4    // example of control of flow
 5    int i;
 6    int isqr(int x) { return x*x; }
 7    for(i=0;i<4;i++)
 8            printf("%d^2 =  %d\n",i,isqr(i));
 9    switch(i) {
10            case 4: printf(" good\n\n"); break;
11            default: printf(" bad\n\n");
12    }
13    // example of some file stuff;
14    // read in some tools
15    #include "tools/nxtString.c"
16    FILE *fp = fopen(_Argv[0],"r");
17    char *p;
18    while((p=nxtString(fp)))
19            printf("%s ",p);
20    fclose(fp);
21    printf("\n\n");
22    // further example of using command line args
23    if(_Argc) { // this is always true
24            int k=0;
25            printf("Processing command line arguments\n");
26            for(k=0;k<_Argc;k++) {
```

```
27                      printf("%s\n",_Argv[k]);
28              }
29      } else
30              printf("OOPS, an internal error has occurred\n");
```

An EiC shell script is interpreted from the top to the bottom. First the code is compiled to bytecode, in its entirety, and then run. After this, control will be parsed to the `main` function if it exists. However, it is not illegal to have a script that does not include the definition of a `main` function. If the EiC directive `:exit`, which is the directive that terminates an EiC interactive session, is present, it will cause the interpretation of the script to halt at the position `:exit` is encounted, and nothing will have happened other than having the code up to `:exit` operator compiled and parsed – but it will not have been executed. Generally, the code for a function is not executed until it is called, see line 8. Command line arguments are passed into to the global variables `_Argc` and `_Argv`, see lines 16 and 23 to 30. For example:

```
% script1.eic  abc 123 -DHELP
```

Implies that:

```
_Argc = 4,                  _Argv[0] = "sript1.eic"
_Argv[1] = "abc"            _Argv[2] = "123"
_Argv[3] = "-DHELP"         _Argv[4] = NULL
```

0.5 Embedding or linking to EiC

To Link against EiC you first need to build the source distribution. Then linking to EiC from aother programs is done by linking against the EiC libraries (`libeic` and `libstdClib`) in `EiC/lib`. In the directory `EiC/main/examples` there is an example program called `embedEiC.c` that links to EiC. Build and run it from the `EiC/main/examples` directory by entering (assuming EiC has been installed in `/usr/local/EiC`):

```
% gcc embedEiC.c -L/usr/local/EiC/lib -leic -lstdClib -lm
% a.out
```

For communicating commands to EiC from another program there are two functions supplied:

```
int EiC_run(int argc, char **argv);
```

and

```
void EiC_parseString(char *command, ...);
```

The `EiC_run` function is used to run C source files. The `EiC_parseString` function is used to pass C or preprocessor commands to EiC via a string, such as:

```
EiC_parseString("#include <stdio.h>");
EiC_parseString("int a = 10,i;");
EiC_parseString("for(i=0;i<a;i++)"
                " printf(\"%%d\\n\",i);");
```

At present the main facility for sharing data between EiC and other applications is via the address operator `@`:

```
int a @ dddd;
```

The above defines `a` to be an integer and is stored at address `dddd`, which must be an integral constant. The constant address `dddd` is not simply an address conjured up. Its purpose is to enable access to data, or even functions, defined in compiled code.

When applied to function definitions, the limitation at this stage is that the function must take void arguments and return void:

```
void foo(void) @ dddd;
```

The above defines `foo` to be a builtin function located at address `dddd`. For example:

```
int foo[5] = {1,2,3,4,5};
void fooey(void) {printf("fooey called\n");}
     ....
EiC_parseString("int foo[5] @ %ld;", (long)foo);
EiC_parseString("void fooey(void) @ %ld;", (long)fooey);
```

Further, `int foo[5] @ 1256;` defines `foo` to be an array of 5 ints mapped at the specified virtual address and the usual pointer safety rules apply; that is, `foo[5];` will be caught as an illegal operation.

Also, you can pass in data to EiC via setting variables and you can get EiC to output data to a file. In a future release of EiC, more facilities are expected to be added for sharing data between EiC and its embedding system.

With respect to `EiC_run`, to run the file "myfile.c" and pass it the command line arguments "hello" and "world", the following sequence of commands would be used.

```
char *argv[] = {"myfile.c", "hello", "world"};
int  argc = sizeof(argv)/sizeof(char*);
EiC_run(argc, argv);
```

0.6 EiC modules

In a nutshell, EiC modules are related groups of EiC/C functions, which get interpreter'd by EiC or builtin to EiC. Therefore, there are basically two types of EiC modules. Interpreter'd code modules and builtin modules (compiled code). It is also possible for compiled code to make calls (callbacks) to interpreter'd code.

One of the nice features of an EiC module, is that once you have a module built you can add it to another EiC distribution by simply copying it into the 'EiC/module' directory and to remove a module you simply remove it from the 'EiC/module' directory – easy as that.

0.7 EiC vs C

Because EiC can be run interactively, it differs from C in several ways. In this section I will outline what is currently missing from EiC and how EiC differs from ISO C.

Although, EiC can parse almost all of the C programming language, right up front it is best to mention what is currently lacking or different:

1. EiC is pointer safe. It detects many classes of memory read and write violations. Also, to help in interfacing compiled library code to EiC, EiC uses the optional pointer-qualifiers `safe` and `unsafe`.

2. Structure bit fields are not supported.

3. While structures and unions can be returned from and passed by value to functions, it is illegal in EiC to pass a structure or a union to a variadic function (that is, a function that takes a variable number of arguments):

```
EiC 1> struct stag {int x; double y[5];} ss;
EiC 2> void foo(const char *fmt, ...);
EiC 3> foo("",ss);
Error: passing a struct/union to variadic function \T{foo}
```

4. The C concept of linkage is not supported. This is because, EiC does not export identifiers to a linker – as does a true C compiler. EiC works from the concept of a single *translation unit*. However, EiC does support the concept of file scope; that is, static extern variables declared in a file are not visible outside that file.

5. EiC does not parse preprocessor numbers, which aren't valid numeric constants; for example, `155.6.8`, which is an extended floating point constants, will cause an error.

6. EiC supports both standard C like comments `/* ... */` and C++ style comments. Also, when EiC is run in script mode, it treats all lines that start with '`#`' and which can't be interpreted as a preprocessor directive as a comment.

7. There are no default type specifiers for function return values. In EiC it is illegal to not explicitly state the return type of a function:

```
foo() { ... }     /*   error: missing return type */
int foo() { ... } /* correct, return type specified */
```

8. In addition to function definitions and declarations with an empty parameter list, EiC only supports prototype declarations and definitions:

```
int foo(); /* Empty parameter list allowed */
int f(value) int value { ... } /* Illegal: old style C */
int f(int); /* Allowed, prototype declaration */
int f(int value); /*Allowed, full prototype declaration */
```

9. EiC does not support trigraph sequences, wide characters or wide strings: nor does it support the standard header `<locale.h>`.

10. EiC's preprocessor lacks the `#line` directive.

11. For convenience, EiC allows the `#include` directive to have an extra form, which permits the parsing of a *token-sequence* in the form `#include filename`; that is, without enclosing double quotes or angled brackets.

12. Besides parsing preprocessor directives or C statements, EiC also parses its own internal house keeping language. House keeping commands are communicated to EiC via lines that begin with a colon.