Extensible Interactive C

Edmond J. Breen

August 16, 2009

Contents

Preface

EiC was developed from a perceived need for a complete interactive C interpreter – or more correctly, an interactive bytecode C compiler that can also run non-interactively in batch mode style. The main advantages of this latter mode is that EiC can compile and run C programs without leaving around executables or object code and that there is no concern about which platform the code is run on.

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 C interpreters built to date. It is suitable as: an aid in teaching C, for fast prototype 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.

However, like EiC, this documentation is also at the beta stage; for example, its library section is still under development and the section on how to `e`xtend EiC, by adding new builtin functionality, has yet to be documented. Therefore, any contributions or suggestions are certainly welcome, and can be sent to me at Ed.Breen@Altavista.net

Copyright

Acknowledgements

Thanks to Martin Gonda for his contribution to EiC's error recovery module; Ross Leon Richardson's for permission to incorporate his online quick reference guide to "The C Standard Library". Hugues Talbot made early suggestions during EiC's development. EiC's type specifier was modeled from lcc, which is available free of charge (Fraser and Hanson, 1995). Part of EiC's runtime library support were derived from the Standard C library, (C), 1992 by P.J. Plauger, published by Prentice-Hall and are used with permission. Thanks to Eugene D. Brooks III for Beta testing EiC and for motivating and supporting many new developments within EiC. In particular: pointer qualifiers, pointer pragma directives and the address specifier `@`. Thanks to Alf Clement for porting EiC to the HP platform. Thanks to Jochen Pohl for porting EiC to NetBSD and for contributing to EiC's makefile system. Jean-Bruno Richard developed EiC's initial ':gen' command and paved the way for getting callbacks to EiC from compiled code working.

Chapter 1

Introduction to EiC

Extensible **i**nteractive **C**, EiC, is a hand crafted, recursive–descent C interpreter. To start EiC simply enter at the system prompt (`%`):

```
% eic
```

As the name implies, EiC is interactive, but see section § 1.2.7, pg: 10 for running EiC in batch mode. Basically the user enters commands, or *immediate statements* at the interpreter prompt `EiC #>`, where the hash mark represents the current line number; for example:

```
EiC 1> #define PI  3.14159
EiC 2> PI * strlen("Hello, World!\n");
```

and then EiC will respond with:

```
43.98226
```

where `strlen`, see § 4.1.13, pg: 132, is a standard C function that will return the number of characters in the argument string `"Hello, World!\n"`:

```
EiC 3> strlen("Hello, World!\n");
        14
```

In fact, virtually all of the C runtime library and a good proportion of the POSIX.1 libray are supported by EiC (but see Chapter 4, pg: 107, for details).

To exit EiC simply enter:

```
EiC 4> :exit
```

EiC is a bytecode (Budd, 1987) compiler that generates its own internal intermediate language known as stack code, which is a bit like the Pascal P-code system (Pemberton and Daniels, 1982). It executes the stack code via its own internal stack machine. The intermediate code produced from the previous call to `strlen` is:

```
0:pushptr  245980          4:stoval
1:bump     1               5:pushint  1
2:checkar  1 1             6:call
3:pushptr  398192          7:halt
```

While the details of the stack code will not be discussed in this document its usage means that executed commands generally perform much faster than an interpreter that uses no intermediate code.

1.1 EiC vs C

Because EiC is interactive 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 (Kernighan and Ritchie, 1988) 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 (see § 3.12.7, pg: 71). To help in interfacing compiled library code to EiC, EiC uses the pointer-qualifiers `safe` and `unsafe`, see § 3.12.5, pg: 67.

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 ‘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*, see § 3.2, pg: 50. However, static global variables remain private to the file they are declared in, see pg: 80

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 (see section § 3.7, pg: 54). Also, when EiC is run in script mode (see § 1.2.7, pg: 11) 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 that permits the parsing of a *token-sequence* in the form `#include filename`; that is, without enclosing double quotes or angled brackets (see section § 2.6, pg: 42).

12. Besides parsing preprocessor directives (Chapter 2, pg: 35) or C statements (Chapter 3, pg: 49), EiC also parses its own internal house keeping language (see section § 1.3.1, pg: 17). House keeping commands are communicated to EiC via lines that begin with a colon.

1.2 Running EiC

To run EiC interactively just enter `eic` at your system prompt:

```
% eic
```

However, you also need to set the environmental variable `HOMEofEiC`, this is so that EiC knows where to find its include files etc. The `HOMEofEiC` environmental variable must be set to point to the directory that contains the EiC include directory. For example: `$HOME/EiC` or `/usr/local/EiC`.

In bash, ksh or zsh use:

```
% export HOMEofEiC=...
```

In tcsh enter:

```
% setenv HOMEofEiC ...
```

where the dots represent the name of the EiC directory

You may wish to include the command in one of your startup scripts such as the `.cshrc` or the `.bashrc` file.

1.2.1 EiC immediate instructions

In interactive mode the user interacts directly with the EiC interpreter. He or She enters C statements, C declarations, preprocessor directives or EiC interpreter commands at EiC's command line prompt:

```
EiC 1>
```

The number before the closing angled bracket represents the current line number.

As a user types out an instruction EiC is analysing the input character stream and is checking for matching brackets. When a closing bracket, either square or curved, is entered, EiC moves the cursor quickly to its matching opening bracket and back again. This can be especially helpful when entering commands with complicated and nested bracketing. However, EiC will produce a beep if an opening bracket cannot be matched to the current closing bracket.

Each immediate instruction produces a type, even if the type is `void`; as for example, C statements, declarations etc. All resulting types and their 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, b;} ab = { 5,3};
        (void)
EiC 7> ab;
        {5,3}
EiC 8> ab.a = 3;
        3
```

While arrays are not expanded, the EiC interpreter display routine considers all `char` pointers to be valid null-character terminated strings (see section § 3.9.4, pg: 58) and will display them as a such. However, as a safe guard against runaway sequences it restricts such print outs to at most 100 characters.

1.2.2 EiC error recovery

It is possible to interrupt the execution of any immediate instruction by pressing control C:

```
EiC 9> while(1);  /* loop forever or until interrupted by <Ctl>C */
EiC interrupted file ::EiC::, line 9
EiC: error clean up entry pt 0
EiC 10>
```

The information displayed between line 9 and 10 is informing that EiC was interrupted at line 9 in file `::EiC::`, which is the name of the interpreter's command line. The next line of information informs that EiC has entered automatic error recovery and garbage collection, or clean up at entry pt 0. The various entry points are of no real concern. Instead, what is interesting is that all new memory allocations, data types, macro definition etc, will be cleaned up and removed. It is EiC's way of trying to set its interpreter back to the state it was before line 9 was entered. In this case there is nothing to clean up but if on line 9 I had entered say `#include foobar.c`, which contained errors, then there could have been potentially thousands of pieces of information to clean up.

While EiC's clean up operation attempts to regain a previous state, it is not always 100% successful; for example, say I have already included the file `foobar.c` then I decide to make some changes to it and then re-include it, but I inadvertently enter a typo, an error, somewhere in the file `foobar.c`. EiC will detect the error and when it is finished translating the entire unit it will trigger the clean up procedure. The clean up operation will then remove all traces of the contents of `foobar.c` from the interpreter and the resulting state will be as if it had never been entered.

1.2.3 Entering multi line commands

EiC has a command line editor where the user can use the delete key and the left or right arrow key to aid editing or one or more of the following commands:

```
printable characters print as themselves (insert not overwrite)
^A moves to the beginning of the line
^B moves back a single character
^E moves to the end of the line
^F moves forward a single character
^K kills from current position to the end of line
^P moves back through history
^N moves forward through history
^H and DEL delete the previous character
^D deletes the current character, or EOF if line is empty
^L/^R redraw line in case it gets trashed
^U kills the entire line
^W kills last word
<LF> and <CR> return the entire line regardless of the cursor position
```

`^A` indicates that the control key is held down while simultaneously pressing the A key, in either upper or lower case.

To input commands into EiC that require multiple lines of code you can just add the backslash '\' character at the end of each line to be continued. For example:

```
EiC 1> double sqr(double x) \
EiC 2> {\
EiC 3>     return x*x;\
EiC 4> }
```

However, as EiC does not supply a full screen editor, a second method for entering multi line commands is to use EiC's C preprocessor to include a sequence of external declarations via file inclusion. For example:

```
EiC 1> #include "examples/sqr.c"
```

and where the contents of `examples/sqr.c` is:

```
#include <math.h>
double sqr(double x)
{
   return x*x;
}
```

1.2.4 EiC on start up

When EiC is launched it automatically looks for a `starteic.h` file. EiC first looks for this file in the current working directory then if this fails, it looks in your home directory and finally if all else fails, it looks in EiC's system `include` directory. It is not a big deal if EiC cannot find a `starteic` file, however the purpose for this file is to allow you to specify defaults on EiC start up.

The default system `starteic.h`, stored in EiC's include directory should resemble:

```
#ifndef _STARTEiCH
#define _STARTEiCH

/* ISO STUFF */
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include <limits.h>
#include <errno.h>
#include <assert.h>

/* POSIX.1 STUFF */
#include <fcntl.h>

#endif /* _STARTEiCH */
```

The start up procedure allows you to define local defaults within a given directory, your own global defaults within your home directory, and finally the EiC system defaults. A possible home directory `starteic.h` might look something like:

```
/* include system headers */
#include <starteic.h>

/* standing macros */
#define sys(x) system(#x)
#define pwd  system("pwd");
#define ls   system("ls");
#define help(x)    system("man 3 " #x)
```

Note, that lines beginning with ':' are EiC command directives, which are explained in detail in section § 1.3.1, pg: 17 and that the home directory `starteic.h` file includes EiC's system `starteic.h`. Therefore, it is assumed that any local `starteic.h` file would inturn include this file.

This facility can also be switched off by passing EiC the `-N` command line switch on start up, see § 1.2.5, pg: 8.

1.2.5 EiC command line switches

You can also change the default behaviour of EiC on start up, by using one or more of EiC's command line swtiches:

```
                EiC
An Extensible Interactive C interpreter
To start eic, type eic.
To exit eic, type :exit.

Usage:
       eic  [-Ipath] [-Dname[=var]] -[hHvVcCrR]  [[file] [fileargs]]
Options:
  C preprocessor directives:
       -Ipath      search for include files in path
       -Dname      define a symbolic name to the value 1
       -Dname=var  define a symbolic name to the value var
                   Note, there is no spaces allowed
  EiC directives:
       -h -H       causes this usage to be displayed
       -v -V       print EiC's Log information
       -p          showline
       -P          show path of include files
       -t -T       turns trace on
       -c -C       turns timer on
       -e          echo HTML mode
       -r          restart EiC. Causes EiC to be re initiated
                      from the contents of EiChist.lst file
       -R          same as 'r', but prompts the user to accept
                      or reject each input line first
       -s -S       run silently
       -f          run in script mode
```

```
-n          no history file
-N          don't use any startup.h files
-A          Non-interactive-mode
file        EiC will execute 'file' and then stop; for example:
              % eic foo.c
fileargs    command line arguments, which get passed onto file
```

This above listing duplicates the response of EiC to:

```
% eic -h
```

The `showline` option is discussed on page 33; the `trace` option on page 28; the preprocessor directives on page 26; the `-P` includes option on page 34; the `-c` option on page 33; the `-v` option on page 34; and the `-N` option is discussed in section § 1.2.4, pg: 7; while the `-r`, `-R`, `-n`, `-A`, `-f`, `file` and `fileargs` options are discussed below.

1.2.6 EiC history file

During an EiC interactive session each command line entered that does not cause an error is saved in a history file `EiChist.lst` in the directory that EiC was launched from; that is, each unique directory used to launch EiC will have its own `EiChist.lst` file. The file is normally created new on each start up and the previous contents (if existing) are ignored. That is, unless the command line switch `-r` is used:

```
% eic -r
```

The switch `-r` used on EiC start up informs EiC to enter its re-initialization mode and the commands stored in the file `EiChist.lst` will be re executed in order of occurrence. The contents of the file `EiChist.lst` is then retained and used to from the start of the history list (discussed on page 27) of the new session. The main purpose for this file is to provided the user with an automatic method for recording an EiC session and provide a way to quickly recapture a previous EiC session. Thus, allowing the user to resume from where he or she left off.

When re-initailizing EiC, it is not always desirable to execute every instruction line in the `EiChist.lst` file. You may wish to edit it first to remove or modify certain lines. To aid in this process it is also possible to re-initialize EiC via the uppercase `R` switch. In this case the user is offered the opportunity to either input the current line, `Y`, to edit the current line before input, `E`, or to not include the current line `N`; for example (where the contents of `EiChist.lst` is `int a, b, c;`):

```
% eic -R
...
Re Initiating EiC -- please wait.
Re-enter [int a, b, c;] (Y/N/E)?
```

The history file mechanism can also be switched off by selecting the `-n` command line option on start up:

```
% eic -n
```

While the following allows EiC to be re-initialized, it prevents EiC from creating a new `EiChist.lst` file:

```
% eic -rn
```

The old file `EiChist.lst` file will be retained and used and no further commands will be added to the list:

1.2.7 EiC non-interactive mode

There are two modes for running EiC and while this document is primarily concerned with EiC's interactive mode, EiC can also be run non-interactively. For example, the following is used to execute the program `examples/hello1.c` in EiC's `examples` directory:

```
% eic examples/hello1.c
```

The above command uses the `file` option that instructs EiC to load the file `hello1.c`, compile it into bytecode, execute it and then to stop. The program, `hello1.c`, is assumed to be a self contained C program, which contains the definition of a `main` function. The `main` function is used to establish the start point of the program:

```
#include <stdio.h>
void message(void)
{
    const char *s = "Hello, world!";
    puts(s);
}

int main(void)
{
    message();
    return 0;
}
```

The entire file, `hello1.c`, plus all it includes is considered to be a single translation unit, see section § 3.2, pg: 50. Also, the default procedure for including `starteic.h` files is ignored and no `EiChist.lst` is utilized. The options for modifying EiC's non-interactive behaviour is limited to the command line options specified in section § 1.2.5, pg: 8.

It is also possible to write programs that take command line arguments in the usual C way, as seen from `examples/main2.c`:

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

The first parameter, which is normally called `argc`, holds the number of argument strings passed to the program and is always at least one. The second parameter, which is normally called `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 examples/main2.c 123 hello -Dworld this.and.that
examples/main2.c
123
hello
-Dworld
this.and.that
```

Running EiC in script mode

In non-interactive mode EiC runs generally like a typical interpreter, accepting input from a complete C program. However, EiC can also run shell scripts non-interactively. For the following examples, in this section only, it will be assumed that you are in EiC's directory `../EiC/module/examples` and that `eic` is installed in `/usr/local/bin`.

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 (see Chapter 2, pg: 35) 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 an 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 byetcode, 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` (as discussed on pg: 23) is present in such a script it will cause it to halt at the position `:exit` is encounted and nothing will happen other than having the code upto `:exit` compiled and parsed but it will not have been executed. Generally, the code for a fuction 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
```

To alter the behaviour of EiC during script-mode just add the appropriate switch to the first line of the script, as seen for `-f` on line 1 above.

1.2.8 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);
```

EiC internet programming

It also possible to run CGI, Common Gateway Interface, scripts via EiC. The function virtualhtml.eic.cgi in directory EiC/module/cgihtml/cgi-bin is used to demonstrate this facility:

```
#!/usr/local/bin/eic -f
:-I/usr/local/EiC/include
#include <stdio.h>
printf("Content-type: text/html\n\n");
printf("<html>\n"
       "<head><title> Simple Virtual HTML </title> </head>\n"
       "<body>\n"
       "<h1>Virtual HTML</h1><hr>\n"
       "Hey look, I just created this page virtually!!!\n");
printf("<p>Date:  %s <br> Current Time: %s\n",__DATE__,__TIME__);
printf("</body></html>\n");
```

Move the function to your cgi-bin and run it from your browser by entering the following line in the browser location window and then press enter:

```
http://www.your_domain/your-cgi-bin/virtualhtml.eic.cgi
```

The output to the browser window, other than a different date and time, should be:

```
Virtual HTML

Hey look, I just created this page virtually!!!

Date: Apr 18 1998
Current Time: 14:25:35
```

You would have noticed on the 2nd line of virtualhtml.cgi the command :-I/usr/local/EiC/include (see page 26 for details on EiC search paths). This is informing EiC to add the directory /usr/local/EiC/include to its search path and is used when looking for include files. The reason this must be explicitly stated in the script is because each CGI script is run in its own shell, which is owned by httpd or www depending on how your webserver is setup and therefore, the HOMEofEiC environmental variable will not have been set (see § 1.2, pg: 3). Note, this instruction is only needed if EiC is not installed in /usr/local or /usr.

EiC debugging CGI scripts

To help debug your C-CGI scripts, EiC provides the command line switch `-e`. It, among other things, tells EiC to inform Netscape as early as possible that the incoming `content-type` is `text/plain`. This turns the browser window into a simple text-like shell which enables output from EiC to be viewed in the usual way:

```
#!/usr/local/bin/eic -fe
#include <stdio.h>
printf("Hello, world wide web!\n");
```

Now, as before, just call the above program, called `www1.eic.cgi`, from Netscape via:

```
http://www.your_domain/your-cgi-bin/www1.eic.cgi
```

The output in your browser's window should be:

```
Hello, world wide web!
```

Note also, if there was a bug or a syntax error in the your cgi-script then EiC diagnostic messages would have quickly pin-pointed the lines of code causing the problem. All the debugging facilities of EiC, such as `trace` (see page 28) and array-bound checking can now be used to help debug your cgi-scripts and get the C part of your script running correctly and within the browser environment.

Running EiC interactively, non-interactively

A further method for running EiC non-interactively that is useful for capturing an interactive session and can be used for reporting errors encounted during such a session is to redirect EiC's standard input to come from a file. The command line switch `-A` is of use here as it instructs EiC that all input from stdin should be treated non-interactively, such as don't bother performing bracket matching etc. As this mode simulates an interactive session all commands must be contained on a single line and an explicit `:exit` must be used to end the session. It differs from EiC's script mode (as discussed above) because each line is compiled into bytecode and run individually; that is, each line is executed as it is encounted. To input commands that require multiple lines, use either the backslash character at the end of each line to continue or use file inclusion. An example of such a script is given in `examples/hello.lst`:

```
#include <stdio.h>
#define str(x)  #x
```

```
#define xstr(x) str(x)
#define W  world!
printf(str(Hello) ", " xstr(W) "\n");
:exit   // DO NOT FORGET TO EXIT
```

And is run via:

```
% eic -As < hello.lst
```

The `-s` switch is used to suppress EiC startup messages. The important point here is not to forget to add the exit directive, `:exit`, to finish execution. Otherwise, EiC will get caught in an infinite loop and you will have to use control Z to pause it and the shell to kill its process.

1.3 The EiC interpreter

In addition to C preprocessor directives (as explained in Chapter 2, pg: 35) the user can enter three main types of input as given by the following grammar[1]:

parse:
: *eic-command parse*
ext-decl parse
stmt parse
DONE

Note, the phases of EiC's translations are discussed in § 3.1, pg: 49.

1.3.1 EiC commands

An interpreter directive as opposed to a C statement, a preprocessor directive, or an external declaration (see below) are communicated from lines beginning with a colon : followed by an *eic-command* production such as:

[1] Appendix A provides an explanation for the notation used to explain the EiC language

eic-command:
  `show` *id*
  `rm` *item*[`,`*item*]*
  `clear` *file-name*[`,`*file-name*]*
  `gen` *header-file* [*num*] [*"outfile"*]
  `exit`
  `variables`
  `variables` [*type-name* | `tags`]
  `help`
  `history`
  `files` [*file-name*]
  `reset` [`here`]
  `-` *comm-switch* [*path*]
  toggle
comm-switch: `one of`
  `I R L`
toggle:
  `trace` [`funcs`]
  `listcode` [`linenums`]
  [`timer` | `showline` | `interpreter` | `memdump` | `verbose` | `includes`]
item: `one of`
  identifier constant-expression
path:
  `any valid directory path`
file-name
  `the name of any included file`

For example, you exit EiC by entering `:exit`. Note, the keywords; that is, the terminal symbols used in the *eic-command* productions will not conflict with C identifiers, as the interpreter distinguishes the difference based upon context of use (see § 3.6, pg: 53). The *eic-command's* are now explained:

show: is used to display type and other information concerning variable and function definitions and declarations. It also provides a quick way to test for the existence of a variable. Example:

```
EiC 1> float (*fval)[5];
        (void)
EiC 2> :show fval
fval -> * ARY[5]float
```

which is read as: fval is a pointer to an array of 5 floats. When `show` is used to display a `structure` or `union` it reveals the size and the members also:

```
EiC 3> struct stag {int x; double da[5];} st;
        (void)
EiC 4> :show st
st -> struct: size 44 bytes
        x -> int
        da -> ARY[5]double
```

If the structure or union contains nested structures or unions, `show` only expands the first level of nesting:

```
EiC 5> struct { float v; struct stag st;} sr;
        (void)
EiC 6> :show sr
sr -> struct: size 48 bytes
        v -> float
        st -> struct: size 44 bytes
                x -> int
                da -> ARY[5]double
```

`Show` can also be used to look at function declarations and definitions:

```
EiC 7> int sqr(int x);
        (void)
EiC 8> :show sqr
sqr -> dec_Func (
        x: int
        ) returning int
        (void)
```

This is interpreted as: `sqr` is a function declaration `dec_Func` that receives an integer argument declared with name `x` and returns an integer to its caller. The prefix `dec_` implies that the body of the function has not yet been defined. Converting the declaration into a definition:

```
EiC 9> int sqr(int x) { return x * x;}
        (void)
EiC 10> :show sqr
sqr -> Func (
        x: int
        ) returning int
        (void)
```

In EiC there are basically two types of functions (see section § 3.18, pg: 86). There are interpreter functions and there are builtin functions. To distinguish these forms the `show` command adds the prefix `Builtin` to builtin functions:

```
EiC 11> :show printf
printf -> Builtin Func (
        ) returning
```

Notice that the argument list is empty and the returning type is undefined. The above informs that while the function `printf`, which is discussed on page 122, is built into EiC, it has not yet been prototyped. No builtin function can be called from EiC until its prototype has been processed:

```
EiC 12> printf("hello\n");
Error in ::EiC:: near line 12: Incorrect function usage: printf
```

However, this is easily rectified by including the appropriate header file:

```
EiC 13> #include <stdio.h>
EiC 14> :show printf
printf -> Builtin Func (
        fmt: * const char ,
        ...
        ) returning int
```

Now, a call can be made to `printf`:

```
EiC 15> printf("hello\n");
hello
        6
```

The 6 is the return value from `printf` and represents the number of characters printed.

The `show` command also helps to promote function documentation: as it displays the first comment `past` the line the opening { bracket of the function is on. Therefore, it provides a simple way of adding function usage. For example, consider the following function stored in `examples/regline.c`:

```
void regline(float *x,float *y, int n,
             float *m,  float *b )
{
    /*  In a least square sense, find the equation:
```

```
 * y = mx + b; Returns the slope in 'm' and
 * the offset in 'b', from the data given in
 * 'y' and 'x'. 'n' being the size of the
 * arrays 'y' and 'x'.
 */
 ...
```

Now from EiC:

```
EiC 16> #include examples/regline.c
EiC 17> :show regline
regline -> Func (
        x: * float ,
        y: * float ,
        n: int ,
        m: * float ,
        b: * float
        ) returning void

    /*  In a least square sense, find the equation:
     * y = mx + b; Returns the slope in 'm' and
     * the offset in 'b', from the data given in
     * 'y' and 'x'. 'n' being the size of the
     * arrays 'y' and 'x'.
     */
```

Because of semantic reasons the comment considered to be the documenting comment will be the *first comment* after the line the opening bracket of the function is on. If the first comment happens to start on the same line as the opening bracket it will not be recognised and the next comment (if it exists) will be used to form the documenting comment. See also the EiC command `listcode` on page 29 for further examples of `show`.

rm: is used to remove mostly variables and functions from EiC's symbol table. Example:

```
EiC 18> :rm fval
        (void)
EiC 19> :show fval
Error in ::EiC:: near line 19: Unknown identifier fval
```

Here the error is simply informing us that the identifier `fval` is no longer recognized by the interpreter.

The operand to the `rm` operator can also be an integral *constant-expression* (see § 3.9, pg: 55). The value of the *constant-expression* is used when removing manually memory leaks as reported by `memdump`. It is an error to attempt to remove a memory item that is not deemed to be a memory leak or to use an invalid item number (see `memdump` page: 32).

clear: is used to remove the contents of entire files from EiC's symbol tables and memory pool. This handy operator removes the contents of an entire file before say re-including it so as to avoid conflicts between variable and function changes that EiC forbids. Example:

```
EiC 1> #include "examples/sqr.c"
        (void)
EiC 2> :clear examples/sqr.c
```

The *file-name*, operand, must match with one of the strings listed by the `:files` operator (page: 24). The `:clear` operator also excepts a comma seperated list of file names, with no white spaces intervening.

gen: The `gen` command takes upto three arguments, *header-file*, *num* and *outfile*:

`:gen` *header-file* [*num*] ["*outfile*"]

The `gen` command is used for generating EiC interfaces to builtin C code. It purpose is to allow the easy interfacing of EiC to libraries of C, and this is covered in more detail in § 5.1.6, pg: 153.

```
EiC > :gen foo.h
```

The above would generate the EiC interface to the *header-file*t `foo.h` to `stdout`.

The *outfile* is used for redirecting the output from `:gen` to a file rather than `stdout`:

```
EiC > :gen foo.h "foo.c"
```

The above would be used to direct the output from `:gen` to the file `foo.c`. Note the *outfile* argument must be a string; that is, enclosed in double quotes..

The *num* option is a constant integer value, and is used to control the level of multiplexing callback code to generate. The default value for *num* is 1:

```
EiC > :gen foo.h 4 "foo.c"
```

exit: is used to terminate an EiC session.

status: is used for inspecting the current status of the EiC toggle switches (see below).

variables: `variables` is like the `show` command (page 18), except rather than showing just one identifier it shows groups of identifiers. However, unlike the `show` command it does not expand structures and unions to reveal their members.

Basically, there are three forms of the `variables` command. When entered on its own it displays all the declared identifiers. Generally, this will supply too much information. Therefore, to limit the information produced by `variables` it is possible to select various subsets by using one of the two other forms:

`variables` [*type-name* | `tags`]

type-name: The *type-name* specifier is discussed in detail in section § 3.22, pg: 91. But briefly it is used to display various identifier types such as, all the pointers to integers:

```
EiC 20> int *p1, *p2, a, b,c;
EiC 21> :variables int *
p1 -> * int
p2 -> * int
```

See the `show` command on page 18 for an explanation of such output.

As an example of using `variables` to view all the functions of a specified form, consider:

```
EiC 22> int f(char * s) { return 1;}
        (void)
EiC 23> int foo(const char * str);
        (void)
EiC 24> :variables  int (char *)
f -> Func (
        s: * char
        ) returning int
foo -> dec_Func (
        str: * const char
        ) returning int
system -> Builtin Func (
        str: * const char
        ) returning int
...
```

Note, *type-qualifiers* are not considered in the matching processing and any matching builtin function will also be displayed.

tags: The `tags` option is used to display the structure, union (§ 3.12.8, pg: 72) and the enumeration (§ 3.12.2, pg: 62) tags that have been declared:

```
EiC 25> struct stag {int x,y;};
EiC 26> enum etag {RED,GREE, BLUE};
EiC 27> :variables tags
etag -> enum
stag -> struct: size 8 bytes
```

help: is used to obtain a quick reference summary of the EiC interpreter commands:

```
EiC 28> :help
-------------------------------------------------------------------------------
EiC-COMMAND          SUMMARY DESCRIPTION
-------------------------------------------------------------------------------
:-I path       Append path to the include-file search list.
:-L            List search paths.
:-R path       Remove path from the include-file search list.
:clear fname   Removes the contents of file fname from EiC.
:exit          Terminates an EiC session.
:files         Display the names of all included files.
:files fname   Summarize the contents of the included file ‘fname’.
:gen fname     Generates EiC interface of the included file ‘fname’.
:gen fname "outfile" Places the interface in outfile
:gen 4 fname   Generates EiC interface with 4 levels of multiplexing.
:help          Display summary of EiC commands.
:history       List the history of all input commands.
:includes      Display path of include files when loaded.
:interpreter   Execute input commands. By default it is on.
:listcode      List stack code.
:listcode linenums     List stack code with associated line numbers.
:memdump       Show potential memory leaks.
:rm  dddd      Remove memory item dddd, which is a constant integer value.
:rm   f        Removes f’s definition from the symbol tables.
:show f        Shows type or  macro definition of ‘f’.
:showline      Show input line after macro expansion.
:status        Display the status of the toggle switches.
:timer         Time in seconds of execution.
:trace         Trace function calls and line numbers during code execution.
:trace funcs   Trace function calls only during code execution.
:variables     Display declared variables and interpreter-ed function names.
:variables tags       Display the tag identifiers.
:variables type-name  Display variables of type ‘type-name’.
:verbose       Suppresses EiC’s copyright and warning messages on start up.
-------------------------------------------------------------------------------
```

files: The `files` command is used to get a list of the names of all the include files currently entered into EiC:

```
EiC 29> :files
::EiC::
```

```
starteic.h
stdio.h
stdarg.h
math.h

...

fcntl.h
sys/fcntl.h
sys/types.h
../doc/regline.c
```

It is also possible to get a summary of the contents of any particular include file:

```
EiC 30> :files fcntl.h
FCNTLH_ -> #define FCNTLH_
open -> Builtin Func (
        path: * const char ,
        access: int ,
        ...
        ) returning int
creat -> Builtin Func (
        path: * const char ,
        amode: usigned short
        ) returning int
        (void)
```

The contents of an include file is summarised by first displaying the declared macros followed by the global variables (if any), which are in turn followed by the function definintions.

reset: The `reset` operator is used to set EiC to a default internal state. All allocated memory is freed, the contents of all include files and global variables, included and declared after the `reset` point, are removed. All previous global scalar variables defined prior to the `reset` point will have their values restored.

The default `reset` point sets EiC to the point that is equivalent to starting EiC with the `-N` command line switch (see, § 1.2.4, pg: 7):

```
EiC 1> :files
::EiC::
starteic.h
```

```
stdio.h
stdarg.h
sys/stdtypes.h
sys/stdio.h
stdlib.h
 ...
        (void)
EiC 2> :reset
        (void)
EiC 3> :files
::EiC::
        (void)
```

It is also possible to define the `reset` point by using the `here` operator:

```
EiC 1> int p = 66;
        (void)
EiC 2> :reset here
        (void)
EiC 3> p = 88;
        88
EiC 4> :reset    // set the reset point to the current state.
        (void)
EiC 5> p;
        66
```

However, there is no guarantee that the reset state will be restored 100%. This is because, EiC, at this stage, will not retore the contents of arrays or structure/union members to their initial values. Also, if a pointer is pointing to some allocated memory that was allocated before the `:reset here` command was issued but freed before the `:reset` command, then the behaviour of that pointer will be undefined.

comm-switch: the *comm-switch* production is analogous to what is commonly known as a C program's command–line switch, which is an argument usually preceded by a dash `-`. Comm-switches are used to modify the behaviour of EiC and its preprocessor. The current valid switches are:

I: insert the given path into the preprocessors search list. Used during file inclusion. Example:

```
EiC 31> : -I./tests
```

Append the directory `tests`, which is off the currently working directory to the search list.

R: remove the given path from the preprocessors search list. Example:

```
EiC 32> :-R ./tests
```

L: list the current search list. Example:

```
EiC 33> :-L
```

The include search list is further discussed in section § 2.6, pg: 42.

history: EiC automatically records each command line as entered from the user in a history list. The default maximum length of the history list is set at compile time and is normally 500 lines. Individual lines are of arbitrary length. When the history list is full old lines are removed from the top while the new command line entries are entered from the bottom.

The user can go backwards through the history list by either pressing the up arrow or by pressing control-p; or forward by pressing the down arrow or control-n. Each line of history can be re-edited and then re-entered by pressing the enter key, `<CR>`. The entire current history list is seen via:

```
EiC 34> :history
float (*fval)[5];
:show fval
struct stag {int x; double da[5];} st;
:show st
struct { float v; struct stag st;} sr;
:show sr
int sqr(int x);
...
:history
```

Note, the list has been truncated manually.

EiC has several keywords that associate with the *toggle* production, § 1.3, pg: 17. They are all toggle-switches that are either turned on or off. That is, they are turned on by entering their command once and turned off by reentering the same command:

```
EiC 1> int i;
        (void)
EiC 2> :timer   // turn timer on
<time taken>: 0
EiC 3> for(i=0;i<100000;++i);
```

```
        (void)
<time taken>: 2.6
EiC 4> :timer  // turn timer off
```

The status of all the toggle-switches can be examined by using the EiC command `status` (pg: 23). The toggle-switches provide EiC with the following optional features:

trace: The trace facility is a toggle-switch (see below) with an extra production. If on, `trace`, traces the function calls and line numbers associated with a given translation unit and prints this information to the screen. Consider the following nonsensical piece of code, which is stored (without the line numbers) in the file `examples/testtrace.c`

```
 1      int f(void)
 2      {   int i;
 3          for(i=0;i<3;++i)
 4              if(i>2)
 5                  break;
 6          return i;
 7      }
 8      int g(void)
 9      {   int k = 0,i = 2;
10          while(i--)
11              k += f();
12          return k;
13      }
14      int main(void)
15      {   int i = 2;
16          do {
17              int k = g();
18          } while(--i);
19          return 0;
20      }
```

Now, `trace` can be used to follow the sequence of program flow:

```
EiC 1> #include examples/testtrace.c
EiC 2> :trace
EiC 3> main();
[main] 15,17,
[g] 9,10,11,
[f] 3,4,3,4,3,4,3,6,
[g] 11,10,11,
[f] 3,4,3,4,3,4,3,6,
```

```
[g] 11,10,12,
[main] 17,18,17,
[g] 9,10,11,
[f] 3,4,3,4,3,4,3,6,
[g] 11,10,11,
[f] 3,4,3,4,3,4,3,6,
[g] 11,10,12,
[main] 17,18,19,
[::EiC::]
```

The first line of the response tells us that control started at function `main` and then passed through lines 15 and 17 after which, control was passed to function `g`. In function `g`, control passed through lines 9, 10 and 11 before entering function `f`, and so on. After leaving function `f`, on line 6, control was passed back to function `g` on line 11, etc. Finally, the trace finished when control was returned back to the EiC command line. The trace facility can also be used during batch mode operations via the command line switch `-t` (see also § 1.2.5, pg: 8):

```
% eic -t examples/testtrace
...
```

Clearly, trace can help in debugging programs: it traces the activation and steps in a sequence of code. It can be used to quickly locate sections of code that are causing crashes or blockages. However, at times, this amount of information can be too verbose, and therefore, the `trace` command has the optional argument `funcs`:

```
trace [funcs]
```

When the extra argument is specified, the trace facility prints out only the names of the functions entered:

```
EiC 4> :trace funcs
EiC 5> g();
[g]
[f]
[g]
[f]
[g]
[::EiC::]
EiC 6> :trace  // turn trace off
```

listcode: `Listcode` will be essentially of interest to those people interested in the bytecode produced by EiC. If `listcode` is toggled on then the bytecode for the current command or translation unit (see § 3.2, pg: 50) will be displayed, non-recursively, to the

screen. That is, it does not show the code for any associated functions. By default, `listcode` is off. For example:

```
EiC 1> int i;
EiC 2> :listcode  // toggle on
   0:halt
        (void)
EiC 3> i = 5;
   0:pushint  5
   1:stoint   0 0
   2:halt
        5
EiC 4> :listcode  // toggle off
```

`Listcode` also affects the output produced from the EiC command `show` (see pg: 18). For example, consider the `swap` function as stored (without the line numbers) in the file examples/swap.c:

```
1       void swap(int *a, int *b)
2       {
3           /* swap the values of a and b */
4           int t = *a;
5           *a = *b;
6           *b = t;
7       }
```

Now from within EiC:

```
EiC 5> #include examples/swap.c
EiC 6> :show swap
swap -> Func (
        a: * int ,
        b: * int
        ) returning void

    /*  swap the values of a and b */

EiC 7> :listcode
   0:halt
        (void)
EiC 8> :show swap
swap -> Func (
```

```
        a: * int ,
        b: * int
        ) returning void

    /*  swap the values of a and b */

   0:checkar  1 0          7:drefint
   1:rvalptr  -1 1         8:refint
   2:drefint               9:rvalptr  -2 1
   3:stoint   0 1         10:bump     1
   4:rvalptr  -1 1        11:rvalint  0 1
   5:bump     1           12:refint
   6:rvalptr  -2 1        13:eicreturn
```

`listcode`, like the `trace` command (page 28) has an extra form: `'listcode linenums'`. When used in this form it displays the associated line numbers corresponding to the bytecode instruction:

```
EiC 9> :listcode linenums  // toggle linenums on
   9:   0:halt

        (void)
EiC 10> :show swap
swap -> Func (
        a: * int ,
        b: * int
        ) returning void

    /*  swap the values of a and b */

   0:   0:checkar  1 0          5:   7:drefint
   4:   1:rvalptr  -1 1         5:   8:refint
   4:   2:drefint               6:   9:rvalptr  -2 1
   4:   3:stoint   0 1          6:  10:bump     1
   5:   4:rvalptr  -1 1         6:  11:rvalint  0 1
   5:   5:bump     1            6:  12:refint
   5:   6:rvalptr  -2 1         7:  13:eicreturn

  10:   0:halt

        (void)
EiC 11> :listcode  // toggle listcode and linenums off
```

Note, any line number with the value zero represents extra "house keeping" code added by the EiC interpreter.

memdump: EiC attempts to keep track of all memory dynamically allocated. If EiC cannot find the owner of a piece of dynamic memory the address and how it was allocated will show up automatically if `memdump` is switched on. For example, consider the following useless piece of code in `examples/leak.c`:

```
1      #include <stdlib.h>
2      void leak(void)
3      {
4          char * s = malloc(10);
5      }
```

The following session provides an example of the usage of memdump:

```
EiC 9> #include examples/leak.c
EiC 10> :memdump   // toggle memdump on
       (void)
EiC 11> leak();
       (void)
item 3656 Create line 917 file stdlib.c nbytes 10
```

The above output informs that the memory item number 3656, which maybe different number during your session of length 10 bytes cannot be associate with an owner. That is, it is a potential leak. Also, it tells us that the memory was allocated from line 917 in file `stdlib.c` rather than from line 4 in file `examples/leak.c`. Strickly speaking, it is correct because if we were to look at line 917 in file `stdlib.c` we would find that this is where EiC does its memory allocation for the EiC interpreter.

It is possible to remove leaked memory items via the `rm` command (page: 21):

```
EiC 12> :rm 3656
       (void)
EiC 13>
```

Also, note from the following session:

```
EiC 1> :memdump  // toggle memdump on
        (void)
EiC 2> char *s = malloc(10);
        (void)
item 3611 Create line 781 file stdlib.c nbytes 10
```

```
EiC 3> free(s);
        (void)
```

shows that all memory allocated dynamically in EiC is considered to be a potential leak – if not freed. This is because EiC does not look at the memory a pointer is pointing to when assigning homes to dynamic memory items. Therefore, the usage of `memdump` should be considered carefully when trying to determine if a genuine memory leak has occured or not. However, `memdump` is still useful as it provides a guide to locating potential memory leaks.

timer: if on, the execution time in seconds of a given translation unit is printed out. By default, the timer is off. From the following piece of code I get from my 66Mhz 486 PC:

```
EiC 1> int i; float f,g;
EiC 2> :timer   // turn timer on
EiC 3> for(i=0;i<200000;++i) f *=g;
        (void)
<time taken>: 8.57
EiC 4> :timer  // turn timer off
```

The `timer` is handy when attempting to optimise

a piece of code as it measures the actual processor time used.

interpreter: if on then input commands will be interpreter-ed. By default it is on.

showline: if on then the input sequence to the interpreter is displayed. Useful for inspecting the expansion of macros. By default it is off:

```
EiC 1> #define help(s)  system("man 3 " #s)
        (void)
EiC 2> :showline   // toggle showline on
        (void)
EiC 3> help(printf);
system("man 3 " "printf");
...
EiC 4> :showline  // toggle showline off
```

This facility can also be turned on from the command line using the switch `-p`:

```
 % eic -p
```

verbose: the verbose command is essentially used when running EiC remotely, as it suppresses EiC's copyright and warning messages on start up. It can also be turned on from the command line using the switch `-v`

```
% eic -v
```

includes: If on, the path of include files will be displayed when loaded. This facility can also be turned on from the command line using the `-P` switch:

```
% eic -P
```

and causes EiC to reveal the paths of all the files it includes.

Chapter 2

The EiC Preprocessor

The EiC preprocessor helps to reduce programming effort and to produce more readable code as it provides a way to associate constants (see§ 3.9, pg: 55) and other text to symbolic names. For example, the following definition:

```
#define PI     3.14159
```

associates the floating point constant 3.14159 to the symbolic name `PI`. When ever the preprocessor sees the name `PI` it automatically replaces it with the text 3.14159. This is very useful, because magic numbers, such as 3.14159, are specified in just one place and can be referred to by name. However, the EiC preprocessor is a lot more powerful than this and all input into EiC is first passed through EiC's preprocessor. It provides for macro substitution, conditional interpretations and file inclusion. While EiC's preprocessor commands are, as much as possible, ISO C compliant and as will be explained below, it lacks:

1. the `#line` directive,
2. and trigraph sequences.

Other than these omissions, as will now be explained, it is a complete C preprocessor.

2.1 Directives

Preprocessor commands are also called directives and program lines beginning with the hash mark `#`, which in turn may be optionally preceded by white space, are interpreted as preprocessor directives. A line consisting solely of `#` is ignored. Each preprocessor line is normally terminated by the end of line character. However, by writing a '\' at the end of a line that line will be continued onto the next by line splicing, which is also know as line

continuation. Otherwise, the preprocessor directive will be formed from all characters up to the end of the current line.

Also, line splicing precedes tokenisation (see below). Lines spliced together will not contain the backslash character and they will continue from the first nonwhite character on the next line. If a line ends in a backslash then the following line will never be treated as a preprocessor directive:

```
#define DF \
#Doug Funny
```

2.2 The Define Directive

A preprocessor directive of the form:

`#define` *identifier token-sequence*

is a macro definition that will cause the given identifier to be replaced by the given token-sequence. Commonly used for manifest constants; that is:

```
#define PI 3.14159
```

N.B., the same identifier can be defined multiple times as long as the token-sequence remains the same. Otherwise, it is an error.

2.2.1 Function Like Macros

A preprocessor directive of the form

`#define` *identifier*(*arg-list*) *token-sequence*

is a macro with arguments; that is:

```
#define max(a,b) ((a) > (b) ?  (a) :  (b))
```

N.B. There can be no space between the *identifier* and the '('. Text inside quotes and or character constants are not expanded. Also note, that the above macro definition for `max` is **unsafe** since it addresses each argument more than once.

It is advisable that arguments in the definition be protected by parentheses:

```
#define prod(x,y)    ((x)*(y))
```

to help avoid certain ambiguities:

```
#include <assert.h>
...
x = prod(2+4,5);    // expands to x = ((2+4)*(5));
assert(x == 30);
...
```

While the macro `assert` is explained in section § 4.1.1, pg: 107 its intention should be clear; that is, it reports an error if its argument resolves to zero.

In a macro call, the number of arguments must match the number of parameters in its definition. However, a parameter lists can actually be empty:

```
#define N() 5
```

A macro can also take an arbitrary statement as an argument:

```
#define insert(stmt) stmt
#define seq10  insert({int i;for(i=0;i<10;i++); A = i;})
    ...
    int A;
    seq10;
    assert(A==10);
    ...
```

When a macro with parameters is invoked there can be whitespace between its name and the left parenthesis.

```
#define BIG    max    ( 0       ,  100 )
assert(BIG == 100);
```

Therefore, a macro call is defined by an *identifier* followed by optional white space followed by `(`, then the parameters, which are followed by `)`. A macro call can also extend across multiple lines without the use of the backslash character:

```
...
assert(7==max(5,
              max(6,
                  7)
              )
       );
...
```

While this feature is all well and good if input is coming from a file, since it allows laying out the code such that visibility is increased - especially if long identifier names are used. However, EiC will refuse to extend a macro call across multiple lines without the presence of the backslash character while running interactively:

```
...
EiC >  max(5,         // Illegal line continuation during macro expansion
```

The reason for this is that with a complex statement the carriage return may be pressed without the user realising that he/she has entered an incomplete macro call; the interpreter won't flag any warnings because it is just expecting more input. Hence, this means that the user can lose synchronisation with the interpreter – and without realising it. Therefore, this feature is made illegal during interactive mode (see also § 1.2.3, pg: 6).

A macro definition can be used to mask a real function by redefining its identifier:

```
int f(int x) { return x*x;}
#define f(x) ((x)+5)
    ...
    assert(f(5) == 10);
    ...
```

However, it is possible to suppress the effect of the macro by enclosing the name of the function in parenthesis. This works because the name of the function/macro is then not followed by a left parenthesis:

```
    ...
    assert((f)(5) == 25);
    ...
```

2.3 The Undef Directive

The `#undef` *identifier* directive is used to remove a previously defined macro definition:

```
#define myerr(x)     fprintf(stderr,"%s",x)
    ...
#undef myerr
```

It is NOT an error to attempt to undef an *identifier* that has not been defined.

2.4 Macro Expansion Rules

The text to be replaced by a macro is first isolated by removing leading and trailing white space. For macros with arguments, all arguments are first collected and then each argument is isolated as just described. Isolated text then undergoes expansion or replacement. After each expansion the resulting text is always rescanned for the occurrence of new macro *identifiers*. This allows for the nesting of macros; for example:

```
#define max3(a,b,c) max(a,max(b,c))
```

will take two expansions to expand:

```
max3(a,b,c)
  max(a,max(b,c))
   ((a) > (((b) > (c) ? (b) : (c))) ? (a) : (((b) > (c) ? (b) : (c))))
```

However, once a given identifier has been replaced in a given expansion it is not replaced if it turns up again during rescanning; instead it is left unchanged:

```
#define char  unsigned char
   ...
    char b = 200;          // 8 bit unsigned data
    assert(b==200);
   ...
#undef char
    char b = 200;      // 8 bit signed data
    assert(b == -56);
    ...
```

Text surrounded by double quotes or characters surrounded by single quotes are never expanded; that is, they are protected:

2.4.1 The Stringization Operator:

A single # preceding a `token` will be recognized by the EiC preprocessor as the ISO C stringization operator. Otherwise a single # symbol will only have significance if it is the first nonwhite character of a line.

The stringization operator influences the replacement process. A `token` preceded by # will cause the preprocessor to replace both # and the `token` by a quoted `token`:

```
#define S(x)  #x
...
S(Hello, world!);  // results in "Hello, world!";
```

During the stringization process, the *token-sequence* is scanned and a backslash character is inserted before each double quote or before each backslash character. Also, all sequences of white space are replaced by a single space character:

```
assert(strcmp("hello world",S(hello world))==0);
assert(strcmp("\"hello world\\n\"",S("hello world\n"))==0);
assert(strcmp("Its a nice day", S(Its   a   nice
  day)) == 0);
```

While there are many usages for the stringization operator, the most common usage is to display variables and C statements:

```
#define ASSERT(x)   if(!(x)) puts("error with: " #x)
   ...
   ASSERT(5 == 7);    // result: error with: 5 == 7
```

2.4.2 The Merging Operator:

An ISO C preprocessor controls the merging of tokens via the merging operator `##`. If a *token sequence* contains `##`, then the text before `##` is merged with the text just after it and the `##` operator along with any white space either side of it is removed:

```
#define cat(x,y)   x ## y
   ...
   cat(Nice,Day);    // results in NiceDay;
```

After each replacement the new token will be rescanned. Also, the `##` operator can not appear at the beginning or end of a *token sequence*. However, a word of warning: the token merging operator can produce non-intuitive output; for example, while `cat(1,2)` creates 12, `cat(1,cat(2,3))` creates `1cat(2,3)`, and this is probably not the original intention. To achieve the effect of nesting `cat` macros you need to be a little more cunning:

```
#define xcat(x,y)  cat(x,y)
    ...
    assert(xcat(xcat(1,2),3) == 123);
    assert(xcat(1,xcat(2,3)) == 123);
```

2.5 Predefined Macros

EiC has complied with the ISO standard and has the following five predefined macros available, and non of which may be redefined:

__LINE__ : resolves to a decimal constant containing the current source-file line number that is being processed. The first line of a file is always 1.

__FILE__ : resolves to a string literal containing the name of the current file being processed.

__DATE__ : resolves to a string literal containing the calendar date, in the form: `Mmm dd yyyy`.

__TIME__ : resolves to a string literal containing the current time, in the form: `hh:mm:ss`.

__STDC__ : resolves to 1.

The __TIME__ and __DATE__ macros are useful for recording compile dates for program versions:

```
printf("build %s, %s",__DATE__,__TIME__);
```

The __FILE__ and __LINE__ macros are useful for producing diagnostic messages:

```
#define ASSERT(x)   if(!(x))\
        fprintf(stderr,"In %s line %d:" #x, __FILE__,__LINE__)
#define location()   printf("at line %d\n",__LINE__)
```

The __STDC__ macro is useful when writing programs that maybe compiled with a non-ISO C compiler.

EiC also has its own specific predefined macros:

_EiC: resolves to 1. Used to isolate EiC specific code within C header files

The following reflects an EiC session:

```
EiC 1> __TIME__;
        17:50:32
EiC 2> __DATE__;
        Jul 21 1996
EiC 3> __STDC__;
        1
EiC 4> __LINE__;
        4
EiC 5> __FILE__;
        ::EiC::
EiC 6> _EiC;
        1
```

Note, the __FILE__ name for the EiC interpreter is "::EiC::".

2.6 The Include Directive

A preprocessor directive of the form

`#include` <*file-name*>

causes the above line to be replaced by the entire contents of the file *file-name*. The file is searched for according to the standard search path list (see also discussion on adding paths to the search list on page 26). A preprocessor directive of the form

`#include` ‘‘*file-name*’’

causes the search to begin first in the current working directory and if this fails it searches for the file in those directories specified via the search path list.

Alternatively, a preprocessor directive of the form

`#include` *token-sequence*

causes first the *token-sequence* to be expanded as is for normal identifier text and strictly ISO style, one of the two forms, < ... > `or "..."`, should result. However, EiC first expands the token-sequence and then treats the resulting text as a *file-name*; that is, a preprocessor directive of the form

`#include` *file-name*

is legal in EiC.

2.7 The Conditional Directive

The Conditional directives provide the preprocessor with the ability to pass or to not pass various lines of text onto the C parser according to the following syntax:

conditional:
    if-line text elif-parts [`#else`] `#endif`
if-line:
    `#if` *constant-expression*
    `#ifdef` *identifier*
    `#ifndef` *identifier*
elif-parts:
    if-line text
    [*elif-parts*]
if-line:
    `#elif` *text*

Like all preprocessor directives, the conditional directives must appear on a line by themselves. The preprocessor constant-expression differs from that of the C language; it must resolve to an integral type and be evaluated at compile time. The syntax for preprocessor's constant-expression is given below (see § 2.10, pg: 46). Also, all *if-line* conditions must be used in conjunction with the `#endif` directive. There is also the optional `#else` directive that can be used to provide an alternative if the initial condition fails. There is also the `#elif` directive for when several alternatives are required (see below).

2.7.1 The `#ifdef` and `#ifndef` directives

The conditional directives `#ifdef` and `#ifndef` are used to cause different parts of a program unit to be translated or not, depending upon whether certain identifiers have be defined or not.

A preprocessor directive of the form

`#ifdef` *identifier*

causes the preprocessor to check to see if the identifier has been defined; i.e. via `#define`. If so, then the directive is said to be fulfilled and all lines of text up to the next `#else` or `#endif` will be processed. Otherwise, these lines will be skipped. Also lines located between an optional `#else` and the `#endif` will only be processed if the `#ifdef` is not fulfilled.

Alternatively, a preprocessor directive of the form

`#ifndef` *identifier*

is only fulfilled if the *identifier* has not been defined.

It is common to use these macros to form a macro, which will ensure that the lines of text between the `#if` and `#endif` are at most considered only once; for example:

```
#ifndef STDIOH_
#define STDIOH_
...       // contents of stdio.h
#endif
```

2.7.2 The `#if` directive

The conditional-directive `#if` also causes different parts of a translation unit to be translated or not, but rather than be depending upon whether or not an *identifier* has been defined, it depends upon the whether that the value of a *constant-expression* is zero or not; for example:

```
#if constant-expression
   .... //pass these lines of text
[
   #else
       ... //pass these lines of text
]
#endif
```

Note: the upright square brackets [ ], are used to denote that the `#else` directive is optional.

2.7.3 The `#elif` directive

The `#elif` is used when several alternatives are required and are evaluated in order until one is satisfied:

```
#if constant-expression
   .... //pass these lines of text
#elif constant-expression
   ... //pass these lines of text
#elif constant-expression
   ... //pass these lines of text
[
   #else
       ... //pass these lines of text
]
#endif
```

2.7.4 The defined operator

The defined operator can be used only in a preprocessor constant-expression. It has the following syntax:

defined-operator:
`defined` *identifier*
`defined (` *identifier* `)`

The defined operator evaluates to 1 if the identifier has been defined, else it evaluates to 0. It has the advantage that it can be used to test for the existence of more than one identifier at a time:

```
#if defined(_EiC) && !defined(UNIX)
...
#endif
```

2.8 The #error directive

Syntax:

```
#error token-sequence
```

The `#error` directive is usually used to flag that a conditional directive has failed:

```
#if OS == SUNOS
   .... //pass these lines of text
#elif OS == SOLARIS
   ... //pass these lines of text
#elif OS == ALPHA
   ... //pass these lines of text
#else
   #error "Unknown operating system"
#endif
```

It causes the preprocessor to output a diagnostic message and which will also undergo normal macro replacement.

2.9 The #pragma directive

Syntax:

```
#pragma token-sequence
#pragma
```

The `#pragma` directive is vaguely defined in ISO C. Its purpose is to permit implementation specific C compiler directives or to add new preprocessor features. For instance, in some implementations it maybe possible to turn on or off certain compiler warning options using a warning `pragma`; for example:

```
#pragma  warning +xxx
#pragma  warning -yyy
```

would specify that the `xxx` warning should be turned on, while the `yyy` warning should be turned off. The problem is that different compilers have different `pragmas` and hence all unrecognised `pragmas` must be simply ignored.

In EiC there are only the pointer `pragmas` and are described in § 3.12.5, pg: 67.

2.10 Syntax of the EiC preprocessor

The grammar parsed by EiC's preprocessor is given below. The productions for *identifier*, *int-const* and *char-const* plus an explanation of the notation used are given in Appendix A.

The preprocessor *constant-expression* is subject to normal macro replacement and after macro expansion, all defined *identifiers* are replaced by the constant 1, otherwise they are replaced by the constant 0. Similarly, but before scanning for macros `defined` *identifier* or `defined (`*identifier*`)` are replaced by the constant 1 if the identifier is defined or by 0 otherwise.

pre-command:
`#define` *identifier token-sequence*
`#define` *identifier*(*arg-list*) *token-sequence*
`#undef` *identifier*
`#include` <*file-name*>
`#include` *"file-name"*
`#include` *token-sequence*
`#error` *token-sequence*
conditional
conditional:
if-line text elif-parts [`#else`] `#endif`
if-line:
`#if` *constant-expression*
`#ifdef` *identifier*
`#ifndef` *identifier*
elif-parts:
if-line text
[*elif-parts*]
if-line:
`#elif` *text*

token-sequence:
[*token-sequence*] `#token` [*token-sequence*]
[*token-sequence*] `token##token` [*token-sequence*]
token-sequence `token`
arg-list:
identifier [*, identifier*]*

constant-expression:
and1-expr [`||` *and1-expr*]*

and1-expr:
or2-expr [`&&` *or2-expr*]*

or2-expr:
    xor-expr [`|` *xor-expr*]*

xor-expr:
    and2-expr [`^` *and2-expr*]*

and2-expr:
    equal-expr [`&` *equal-expr*]*

equal-expr:
    rel-expr [`==` *rel-expr*]*

rel-expr:
    shift-expr [[`<` `<=` `>=` `>`] *shift-expr*]*

shift-expr:
    ar1-expr [[`<<` `>>`] *ar1-expr*]*

ar1-expr:
    ar2-expr [[`+` `-`] *ar2-expr*]*

ar2-expr:
    primary-expr [[`*` `%` `/`] *primary-expr*]*

primary-expr:
    (*constant-expression*)
    int-const
    char-const
    identifier
    [*!* + - ~] *primary-expr*
    defined-operator
defined-operator:
    `defined` *identifier*
    `defined (` *identifier*)

Chapter 3

EiC's C Specifications

EiC has been hand coded. Its parsing method is LL(N) and its grammar was derived from the LR grammar presented in Appendix A of (Kernighan and Ritchie, 1988). Here the EiC programming language and specifications is given as this will, hopefully, allow for future developments. The syntax notation used when specifying the grammar for the EiC and C language is described in section § A.1, pg: 161.

3.1 Phases of translation

The input program text is translated by EiC in logically successive phases:

1. Program lines ending in \ are extended onto the next line.

2. Comments are stripped out and are replaced by a single space.

3. The input sequence is then tokenized and any embedded EiC commands will be carried out inplace and in sequence of occurrence. For example:

```
#define foo  xx
:show foo
```

   The preprocessor `#define` directive will be processed before the EiC command `show`, which will be processed before the macro `foo` is expanded. The above EiC show command will result in:

```
foo -> #define foo  xx
```

which is how EiC specifies that foo is a macro (see the `show` command on page 18).

4. Any preprocessor directives are next obeyed and macros will be expanded.
5. Escape sequences, character constants and string constants are next recognized and adjacent string constants separated only by white space are concatenated together.
6. The tokenized input sequence is next translated into byte-code to be executed either directly or to be stored as function code for linkage to other translation units.

3.2 Translation units

In C, a translation unit consists of one or more definitions or declarations. In EiC, a unit of input at the EiC command line prompt is consider to be a translation unit. It may consist of one or more declarations, function definition or immediate statements. Include files and all they include are generally considered to be part of the same translation unit. Unless an error has occurred during translation, all translation units will cause execution to occur – even if the execution consist solely of a single `halt` byte-code instruction.

Because EiC can run interactively its definition of a translation unit is weaker than that given for ISO C. With implications that all identifiers defined at level 'file scope' with either explicit or implicit external scope are visible to other translation units and hence from the EiC command line.

When EiC is used to run programs; such as `foo.c`:

```
% eic foo.c
```

foo must contain the definition of a `main` function, which is used to establish the start point of the program. The entire file, `foo.c`, plus all it includes is considered to be a single translation unit.

In EiC, program modules are linked/brought together to form a larger unit using the preprocessor `#include` directive (section § 2.6, pg: 42). See also running EiC non-interactively § 1.2.7, pg: 10.

3.3 Tokens

A token is a sequence of non-white characters having a collective meaning. The characters separating the tokens are collectively know as white space, which is composed from: spaces, tabs, newlines, form feed and or comments. In general, there is a set of strings in the input for which the same token is produced, as shown in Table 3.3:

Table 3.1: Token examples

Token	string
`for-sym`	for
`if-sym`	if
`float-const`	0.5, 1.0E-20, 1e-3, etc
`char-const`	'a', 'z', '\377', etc

EiC recognizes five major groups of tokens: identifiers (usually abbreviated to id), keywords, constants (character, string and numeric), operators and punctuation marks.

White space is usually stripped out of the input stream. However, if the input is coming from the keyboard then the newline character is significant and is replaced by the terminal `DONE`. This is equivalent to the end of file mark, `EOF`, when reading input from a file.

3.4 Identifiers

An identifier is a name that is formed from a sequence of letters, digits and underscores. The first character of an identifier must be a letter or an underscore character. In EiC, syntax is case sensitive and therefore, upper and lower case letters are different. There is no restriction on the length of an identifier.

id:
  letter [*letter, digit,* _]*
  _ [*letter, digit,* _]*
letter: `one of`
  `a b c d e f g h i j k l m n o p q r s t u v w z y z`
  `A B C D E F G H I J K L M N O P Q R S T U V W Z Y Z`
digit: `one of`
  `0 1 2 3 4 5 6 7 8 9`

3.4.1 Identifier restrictions

The following restrictions are placed on identifier names.

1. An identifier name cannot be the same as a keyword (§ 3.8, pg: 54).
2. All library function identifier names are reserved at all times.
3. All identifiers that begin with an underscore should be considered reserved.

4. All identifiers that begin with `EiC_` or `eic_` are considered reserved for EiC's future developments.

5. All identifiers beginning with `is`, `to`, `mem`, or `str` and followed by another lower case letter are considered reserved for ISO C future library implementations.

6. While there is no restriction on the length of an identifier name in EiC, ISO C only requires that the first six characters of each identifier be unique.

Note, only the first restriction is enforced by EiC – like most other C implementations. Therefore, it is up-to the programmer to enforce and beware of these other rules and limitations.

3.5 Scope Rules

Scope rules determine how references to non-local names are handled and the visibility of local names. In EiC, like ISO C and many other languages such as Pascal and Ada the lexical scope rule is used. That is, the declaration of a name is handled simply by examing the program text. The scope of an identifier relates to the portion of program text in which the identifier is active. The same identifier may be used at different scope levels for different purposes. The scope of a variable lasts until the end of the block in which it is declared in. In EiC, blocks are delimited by braces { and }, and have the general form:

{ *declaration-list*$_{op}$ *statements*$_{op}$ }

Blocks may appear as translation units or anywhere *statements* can. Delimiters ensure that one block is either totally separate from another block or is totally nested within another block. Therefore, it is a simple matter to assign a scope level to a block and to the identifiers assigned with it. For example: the scope level outside any block or function parameter–type–list is 1. From the listing below, the identifier `x` declared on line 1 is outside any block; therefore, EiC will automatically assign it to level 'file scope', 1. This variable is active only on those lines that end with `/* scope level 1 */`. This is because it gets masked by the identifier `x` declared on line 5 and is active only on the lines ending with `/* scope level 2 */`. The `x` identifier declared on line 5 is also masked by the next identifier `x` declared on line 8. This latter identifier is active only at scope level 3. In contrast, identifier `y` declared on line 1 is active or visible from the location of its declaration to the end of the listing.

```
1       int x,y;        /* scope level 1 */
2       ...             /* scope level 1 */
```

```
 3      int foo(void)   /* scope level 1, parmater list at level 2 */
 4      {               /* scope level 2 */
 5          float x;    /* scope level 2 */
 6          ...         /* scope level 2 */
 7          {           /* scope level 3 */
 8              int x;  /* scope level 3 */
 9              ...     /* scope level 3 */
10          }           /* scope level 2 */
11          ...         /* scope level 2 */
12      }               /* scope level 1 */
13     ...             /* scope level 1 */
```

Thus, a block can be seen as a form of a name-less, parameter-less function. However, functions, unlike blocks, can not be nested.

Note: macro definitions are scope less. Macros are always visible and can never be masked by another identifier or macro. However, a macro can mask an identifier.

3.6 Name Space

In C, identifiers are grouped into at least four name spaces: **1)** variables names, **2)** structure, union and enumeration tag names, **3)** labels for goto statements and **4)** structures and unions have their own area for member names. However, functions don't have a separate name space for parameters or local variables; these variables are handled using scope rules. Also, the same identifier can exist in different name spaces without causing conflicts as shown by the following:

```
struct node {
    int node;
} node;

node:
    goto node;
```

The first occurrence of the identifier `node` is entered into the tag-name space. The second occurrence is entered into the structure-member name space for the structure node, which in turn is entered into the variable-name space. The fourth occurence of `node` is entered into name space for labels.

Also, in EiC, there is a separate name space for EiC command identifiers. Having a separate name space for EiC command names allows the use of these names to be overloaded as identifiers for objects and functions:

```
EiC > int show;
        (void)
EiC > :show show
show -> int
        (void)
```

Hence, the eic-command `show` does not get confused with the identifier `show`.

3.7 Comments

Under standard conditions, EiC just allows two styles of comments: the traditional ISO C style comment `/*...*/` plus the C++ style of comments. A sequence of characters in the input stream beginning with `/*` and ending with `*/` or beginning with `//` and ending at the end of the current source line constitutes a comment in EiC. The traditional style of comment `/*...*/` cannot be nested but because the `//` is not recognized inside the traditional comment, nesting can be indirectly achieved via:

```
/*
    int any;   // this number will hold anything
    ...
*/
```

However, the `#if 0 ... #endif` construct is often a more preferable way of disabling large sections of code from being translated.

EiC strips out all standard comments automatically from the input stream before preprocessing and they are replaced by a single space character; for example `a/* a comment */b` will be replaced by `a b`. Comments are not recognized within quotation marks; that is, a string literal (see § 3.9.4, pg: 58); nor within character literals (see § 3.9.3, pg: 57). However, both string and character literals can occur within a comment.

Also, when EiC is run in script mode (see § 1.2.7, pg: 11) it treats all lines that start with '`#`' and which can't be interpreted as a preprocessor directive as a comment.

3.8 Keywords

The following identifiers are reserved:

```
auto      break     case          char
const     continue  default       do
double    else      enum          extern
float     for       goto          if
int       long      register      return
short     safe      signed        sizeof
static    struct    switch        typedef
union     unsafe    unsigned      void
volatile  while     __eiclongjmp  __eicsetjmp
```

3.9 Constants

The following constants are recognized by EiC:

constant:
int-const
float-const
char-const
string-const

Also, the value of a numeric constant is always positive. Any minus sign present is not considered part of the constant: it is part of an unary expression and not of the constant itself. Below are descriptions and the syntax for the various literal classes recognized by EiC.

3.9.1 Integer Constants

An integer constant consists of a sequence of *digit* types. If the sequence begins with zero then the input number is expected to be in hexadecimal or octal format depending on whether the next input character is the letter `x`, `X` or a digit. Hexadecimal numbers also include the letters `a` or `A` to `f` or `F`.

Immediately following the numeric part of an integer literal can be the optional integer suffix, which consists of `u` or `U` to indicate that the number is unsigned and/or `l` or `L` to indicate that the number is long.

The type of an integer literal whether it is `long` or `unsigned` will depend on its form and suffix. Unless otherwise specified, if an `int` can represent the value of the original type then it will be converted to an `int`; A decimal constant whose value exceeds the largest signed machine integer is taken to be `long` or `unsigned long`, which ever fits first. Likewise, an octal or hex constant that exceeds the largest signed machine integer is likewise taken as an `unsigned int`, a `long` or an `unsigned long`, which ever fits first.

int-const:
    *nonzero-digit digit** [*int-suffix*]
    `0` *hex-octal-const* [*int-suffix*]
hex-octal-const:
    hex-const
    octal-const
octal-const:
    *octal-digit**

hex-const:
    [`x`, `X`] *hex-digit**
int-suffix:
    long-suffix [*unsigned-suffix*]
    unsigned-suffix [*long-suffix*]
long-suffix: `one of`
    `l L`

unsigned-suffix: `one of`
    `u U`
hex-digit: `one of`
    digit `A B C D E F a b c d e f`
octal-digit: `one of`
    `0 1 2 3 4 5 6 7`
nonzero-digit: `one of`
    `1 2 3 4 5 6 7 8 9`
digit: `one of`
    `0 1 2 3 4 5 6 7 8 9`

The limits on integer constants are stored in the header file `limits.h`

3.9.2 Floating Point Constants

A floating–point constant may consist of a decimal point, an exponent or both. A *float-suffix* can be used to specify the type of the constant: `f` or `F` for `float`; `l` or `L` for `long double`, which in EiC is identical to `double`. Unless specified, a floating-point constant will be of type `double`:

float-const:
    digit-seq f-float-const
    `.` *digit-seq* [*exp*] [*float-suffix*]
f-float-const:
    `.` [*digit-seq*] [*exp*] [*float-suffix*]
    exp [*float-suffix*]
    float-suffix
float-suffix: `one of`
    `f F l L`
exp:
    [`e`, `E`] [*sign*] *digit-seq*
sign: `one of`
    `+ -`
digit-seq:
    [*digit*]$^+$

The floating point limits are stored in the header `float.h`.

3.9.3 Character Constants

Character constants are represented by one or more characters enclosed in single quotes; such as, `'\n'` or `'n'`. Character constants are of type `int`. The value of a single character constant is its ASCII value; for example, `'A'` == 65 and `'B'` == 66. The following set of control characters is recognized:

new line	\n	backslash	\\
horizontal tab	\t	question mark	\?
veritcal tab	\v	single quote	\'
backspace	\b	double quote	\''
carriage return	\r	octal number	*000*
fromfeed	\f	hex number	\x*hh*
bell	\a	hex number	\X*hh*

As seen from the above table, octal and hexidecimal numbers can also be used to form character constants; for example, `'\377'` == `'\xff'` == -1:

char-const:
'*c-char*$^+$'
c-char:
`any character other than a single quote or backslash`
escape-sequence
escape-sequence: `one of`
`\n \t \b \r \f \v`
`\\ \' \'' \a \?` *\0*
\00 \000 \xh \xhh \xhhh
\Xh \Xhh \Xhhh
h:
hex-digit
0:
octal-digit

ISO C allows for an extended character set or a wide character set; that is, characters that cannot be represented by the `char` type. EiC does not recognize this set of characters.

3.9.4 String Constants

A string constant is possibly a zero length array of characters enclosed in double quotes. Its type and storage is initially `static char []` that eventually gets cast to `char *` and unless it is used as an argument to the `sizeof` operator. Its syntax is:

string-const:
"[*s-char*]*"
s-char:
`any character except the double quote, backslash or newline`
escape-sequence

The double quote, backslash or newline characters are included into string constants by using the escape code mechanism:

```
printf("%s %s","hello","\"hello\"");
```

prints:

```
hello "hello"
```

Adjacent string constants separated only by white space are concatenated prior to parsing; this is a handy feature as it makes it easy to construct formated output with added efficiency of just a single function call:

```
puts("this is line 1\n"
     "this is line 2");
```

A string can also be continued via the use of the backslash character \:

```
puts("this is line 1 \
      this is also line 1");
```

The output from the above call to `puts` shows that the backslash and the newline characters are ignored but that the whitespace on the continuation line is not:

```
this is line 1       this is also line 1
```

String constants are stored in a null terminated sequential block of characters as seen for the string "Hello, world!":

H	e	l	l	o	,		w	o	r	l	d	!	0

String constants can be fed into the `sizeof` operator, which will return the number of characters spaces assigned to the array; For example `sizeof("Hello, world!");` returns 14 and not 13 the number of characters in the array. String constants can be used to initialize an array of characters or pointers to characters:

```
char str[] = "this is an array of characters";
char *pstr = "this is a pointer to an array of characters";
```

Note: `sizeof(str)` = 31, while `sizeof(pstr)` = 4, which is the size of a pointer on my system at the time of writing this document.

Wide strings: EiC does not recognize `wide strings`; that is, a string constants prefixed with the letter `L`.

3.10 External declaration

In the following sections the notion for a definition and a declaration will be presented. As a word of introduction, a definition is a declaration that reserves storage for a given C object – otherwise the declaration is just a reference symbol.

C's external declaration *ext-decl* consists of a sequence of external declarations that can be either a C declaration, *declaration*, or a function definition, *func-def*; that is:

ext-decl:
    declaration
    func-def

The discussion of function definitions will be deferred till § 3.17, pg: 84.

3.11 Declarations

EiC's syntax for the C declaration has the form:

decl:
    decl-spec [*init-decl-list*]
decl-spec:
    storage-class [*decl-spec*]
    type-spec [*decl-spec*]
    type-qual [*decl-spec*]
init-decl-list:
    init-decl [, *init-decl-list*]*
init-decl:
    declarator [= *initialiser*]

A C declaration *decl* begins with one or more specifiers, *decl-spec*, in any order and is normally followed by the optional initial declarator list *init-decl-list*. For example, consider the following declaration for the identifier `cd`:

$$\underbrace{\overbrace{\texttt{extern}}^{storage-class}\ \overbrace{\texttt{long int unsigned}}^{type-spec}\ \overbrace{\texttt{const}}^{type-qual}}_{decl-spec}\ \underbrace{\overbrace{\texttt{cd[2]}}^{declarator}\ = \overbrace{\{25,55\}}^{initialiser}}_{init-decl-list};$$

Because of the vast number of data types, constructors and semantics associated with a C declaration, declarations are by far the most difficult part of the C programming language to parse. Furthermore, a re-declaration of an identifier is NOT illegal as long as both declarations remain compatible:

```
int i,i;                    // okay, compatible declarations
extern float a[], a[10];    // okay, 2nd declaration adds more info
int i; float i;             // error: Ambiguous re-declaration for i
```

However, according to ISO standards, the re-definition of a function is illegal. If EiC was to adopt this recommendation, as is, EiC users would have to laboriously remove all the function definitions from the scope of the EiC interpreter (see § 1.3.1, pg: 21) before re-including a source file after each editing task. For example, consider the source file `foobar.c`, which contains only the function definitions for `foo1` and `foo2`:

```
EiC> #include foobar.c  // first time
EiC> :rm foo1, foo2  // Now, remove foo1 and foo2 from EiC
EiC> #include foobar.c  // Next, include foobar.c a 2nd time
```

This is obviously problematic, since all the functions defined in `foobar.c` must be specified for their removal. Therefore, if all function definitions remain compatible (i.e., have the same name, return the same type and accept the same arguments) EiC will simply warn about each functions redefinition:

```
EiC> #include foobar.c // first time
EiC> #include foobar.c // 2nd time
Warning: in foobar.c near line 5:  Function Re-definition of foo1
Warning: in foobar.c near line 20: Function Re-definition of foo2
```

Note, that with the preprocessor directive `#include`, the file name was not surrounded by quotes or angled brackets, see § 2.6, pg: 42 for an explanation.

3.12 Type specifiers

The C language provides a large number of built in types. Type specifiers attribute various properties to a C object. There are the following basic data types:

type-spec: `one of`
`void char short int`
`long float double`
`signed unsigned`
enum-spec
struct-or-union
typedef-name

3.12.1 char, short, int and long specifiers

The `char, short, int and long` specifiers form what are known as the integral types. A `char` or `short` may be used in place of an integer and in all cases they will be automatically cast to an integer.

The integral types all have essentially different word lengths and they are signed types; that is, their values by default will range from negative to positive. The `int` specifier in particular is very rubbery. It size can vary in number of bytes and this length will be machine specific. The Table below provides some basic information regarding EiC integral types:

Type Specifier	length in bytes	Range	Example
`char`	1	-128 to 127	`char x;`
`unsigned char`	1	0 to 255	`unsigned char x;`
`short`	2	-32768 to 32767	`short x;` or `short int x;`
`unsigned short`	2	0 to 65535	`unsigned short x;`
`int`	4	-2147483648 to 2147483647	`int x;`
`unsigned int`	4	0 to 4294967295	`unsigned x;`
`long`	4	-2147483648 to 2147483647	`long x;` or `long int x;`
`unsigned long`	4	0 to 4294967295	`unsigned long x;`

Fortunately these limits are specified in the standard C library header file `"limits.h"` (see § 4.1.5, pg: 110):

```
#include <stdio.h>
#include <limits.h>
int main(void)
{
   printf(" CHAR_MIN = %12d,  CHAR_MAX = %12d\n", CHAR_MIN,CHAR_MAX);
   printf(" SHRT_MIN = %12d,  SHRT_MAX = %12d\n", SHRT_MIN,SHRT_MAX);
   printf("  INT_MIN = %12d,   INT_MAX = %12d\n",  INT_MIN, INT_MAX);
   printf(" LONG_MIN = %12ld, LONG_MAX = %12ld\n",LONG_MIN,LONG_MAX);
   printf("UCHAR_MAX = %12d\n",UCHAR_MAX);
   printf("USHRT_MAX = %12d\n",USHRT_MAX);
   printf(" UINT_MAX = %12u\n", UINT_MAX);
   printf(" LONG_MAX = %12d\n", LONG_MAX);
   return 0;
}
```

3.12.2 The enum type specifier

The enumeration specifier allows for the definition of a set of constant integer values to be easily associated with a set of names. The syntax for the enumeration specifier is:

enum-spec:
    `enum` [*id*] {*enum-list*}
    `enum` *id*
enum-list:
    enumerator
    enum-list `,` *enumerator*
enumerator:
    id
    id `=` *const-expr*

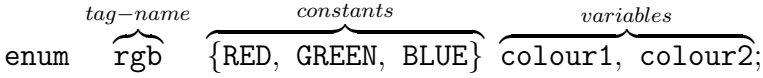
and from which it is easy to see a similarity with the structure or union specifier (see: § 3.12.8, pg: 72):

```
enum  {RED, GREEN, BLUE};
```

The enumeration specifier associates the manifest constants `RED, GREEN` and `BLUE` with the values `0, 1`, and `2` respectively. They are automatically assigned values sequentially starting from zero. The intention here is to make more readable and more easy to produce code than via the preprocessor `#define` directive (§ 2.2, pg: 36):

```
#define  RED     0
#define  GREEN   1
#define  BLUE    2
```

There is also the optional enumeration tag name and enumeration variables:



where the intention is that the tag name will be used to define new variables and the variables will only be assigned the values `RED, GREEN,` or `BLUE`:

```
enum rgb mycolour, yourcolour;
mycolour = RED;
yourcolour = BLUE;
```

It is also possible to initialize the enumeration constants to predefined values using a constant expression of integral type:

```
enum { RED, GREEN = 13, BLUE };
```

Now, `RED` will be assigned 0, `GREEN` 13 and `BLUE` 14.

In EiC the enumeration constants are treated as normal integer constant and enumeration variables are treated as plain integers. Further, EiC performs no type checking to prevent enumerated types and integers types from mixing. Anywhere an integer expression can be used an enumeration constant or enumeration variable can also be used.

Enumeration tag names occupy a different name space than do normal variable names. Therefore, such identifiers can be used at the same scope level for other objects without causing conflicts – although such practices lead to obscure code it is legal:

```
enum rgb mycolour, yourcolour;
int rgb;  /* okay, different name space */
```

At the same scope level, enumeration constant and variables names must all be unique and an enumeration constant or variables within an inner block can mask declarations defined in outer blocks:

```
...
int RED = 5;
{
    enum {RED = 0, GREEN, BLUE};
    assert(RED == 0};
}
assert(RED == 5);
```

3.12.3 float and double specifiers

The specifiers `float` and `double` form the floating point objects. ISO C supports three types of floating point objects: `float`, `double` and `long double`. EiC handles `long double`s as ordinary `double`s.

The `float` specifier is a single-precision floating point number, while the `double` specifier is a double-precision floating point number. Floating point objects are always signed and they have fractional and exponent parts. Scientific notation for floating types is used: 2.22e5 represents the value 222000, where the 'e' or 'E' notation indicates how many positions to move the decimal point left or right depending on the sign of the exponent; for example, 2.22e-5 represents the number 0.000022. All floating point values are stored in normalized form. For example, 0.000123 wastes three zeros on the left of the number that has no meaning except to indicate the position of the decimal point. Normalizing this number gives 1.23e-4 and can be represent by M $\times$ b^k, where M represents the mantissa or significand, b^k the exponent and b the radix. However, in C a more extensive model is used to represent normalized floating point values (see section § 4.1.4, pg: 109).

The floating point equivalent to `"limits.h"` is the standard C header file `"float.h"`, which provides for all the values that characterize the floating-point types. But as a quick, but non-exhaustive, summary: in EiC the floating point objects have the following specification:

Type Specifier	length in bytes	Range	Example
`float`	4	1.175494E-38 to 3.402823E+38	`float x;`
`double`	8	2.225074E-308 to 1.797693E+308	`double x;`
`long double`	8	2.225074E-308 to 1.797693E+308	`long double x;`

and can be verified via the following code:

```
#include <stdio.h>
#include <float.h>
int main(void)
{
   printf(" FLT_MIN = %E,  FLT_MAX = %E\n",FLT_MIN, FLT_MAX);
```

```
    printf(" DBL_MIN = %E,  DBL_MAX = %E\n",DBL_MIN, DBL_MAX);
    printf("LDBL_MIN = %E, LDBL_MAX = %E\n",LDBL_MIN,LDBL_MAX);
    return 0;
}
```

3.12.4 Pointer types

Syntax:

pointer:
    `*` [*pointer-qual-list*]
    `*` [*pointer-qual-list*] *pointer*
pointer-qual-list:
    type-qual-list [*pointer-qual*]
    pointer-qual [*type-qual-list*]
type-qual-list:
    type-qual
    type-qual-list type-qual
pointer-qual: `one of`
    `safe unsafe`
type-qual: `one of`
    `volatile const`

Addresses that are stored in memory are called pointers and the general concept is simple: a pointer is an integral value containing the address of some other object. It specifies the memory location where the data associated with the object can be found. Some people claim that once you have mastered pointers you have mastered C; this is clearly an oversimplification, but does highlight the importance of understanding pointers. The declaration:

```
int *p;   /* p is a pointer to an integer */
```

declares `p` to be a pointer to an integer.

When working with object addresses, the two important operators are: the address operator `&` and the indirection operator `*`. They are the inverse of each other; that is:

```
assert(p == *&p);
```

As seen above, the indirection operator when used in a declaration specifies that the identifier is a pointer. The number of `*` used in a declaration determines the level of indirection; for example, to declare a pointer to a pointer to an int, the token `*`, must be used twice (in EiC, there is no limit to the number of indirections that can be applied):

```
int **q;
```

The `&` can be applied to only variables and array elements. If `x` is an integer then we can assign the address of `x` to `p`:

```
p = &x;
```

Now the indirection operator can be used to obtain the value stored at `x`, via `p`:

```
assert(*p == x);
```

In general, if `p` points to object `x` then `*p` can appear anywhere it is legal for object `x` to appear.

3.12.5 Pointer Qualifiers

EiC has `safe` and `unsafe` pointer qualifiers. Pointer qualifiers are designed to allow the creation of interface routines to embedded C code, which accept as arguments: arrays of pointers or structures that have pointer members. For example:

```
int * safe p;
```

defines `p` to be a `safe` pointer and is the standard pointer type in EiC. The default pointer-qualifier type can be controlled by the use of EiC's `#pragma` directive, see § 3.12.5, pg: 67. The following:

```
int * unsafe p;
```

defines `p` to be an `unsafe` pointer, which is the standard pointer type in C.

In EiC, all pointers are handled as `safe` much like all floats are handled as `doubles` and `chars` are handled as `ints`. Safe pointers are stored with lower and upper bound information that specify a range of legal values, see also § 3.12.7, pg: 71. Thus, the storage requirement is obviously greater for a `safe` pointer than an unsafe pointer.

The following rules apply to EiC pointers. It is illegal to cast between `safe` an `unsafe` pointer addresses:

```
EiC 1> int * * safe p;
EiC 2> int * * unsafe q;
EiC 3> p = q;
Error in ::EiC:: near line 3: Casting between safe and unsafe address
```

This is because the storage requirement of a `safe` pointer is different to that required for an `unsafe` pointer. In C, casts are freely allowed between any object pointer and a `void` pointer. For example:

```
EiC 1> int ***p, **d; void *q;
EiC 2> p = q;
EiC 3> p = d;
Warning: in ::EiC:: near line 3: Suspicious pointer conversion
```

Therefore, it is legal to cast an `usafe` or a `safe` pointer to and from any `void` pointer. However, the bounds of a `safe` pointer will be lost via casting it to an `unsafe` pointer and back again. It is also legal to make casts between a `safe` and an `unsafe` pointer:

```
EiC 1> int * safe p;
EiC 2> int * unsafe q;
EiC 3> p = q;
```

A `safe` pointer is converted to an `unsafe` pointer by discarding the additional safe-pointer information and this is done when the `unsafe` pointer is written to memory. Likewise, an `unsafe` pointer is converted to a `safe` pointer by setting the safe pointer's lower and upper bound values to zero and infinity repectively. Note, casting an `unsafe` pointer to a `safe` pointer does not create a `safe` pointer.

Pointer Pragmas

The default pointer type qualifier can be controlled via the use of three pragmas that work on a stack principle:

```
#pragma  push_safeptr   // default pointer type will be safe
#pragma  push_unsafeptr // default pointer type will be unsafe
#pragma  pop_ptr        // return to previous pointer type
```

The default pointer state in EiC is safe; for example:

```
EiC 1> int *p;                  // defines a safe pointer
EiC 2> #pragma push_unsafeptr
EiC 3> int *q                   // defines an unsafe pointer
EiC 4> #pragma pop_ptr          // return to previous pointer state
```

Pointer Arithmetic

If object `x` is of size `s` in bytes then adding or subtracting the integral value `i` to `p` (i.e., `p = p` $\pm$ `i`) causes `i` $\times$ `s` to be added or subtracted to or from the value stored at `p`.

The relationship between arrays and pointers are easily seen via an example:

```
int *p, a[100];
p = &a[0];    /* point to the beginning of the block */
assert(*(p+20) == a[20]);
```

Note: In C, if the resulting pointer, `p+x`, points outside the bounds of a legal array, except for the first location beyond the end of the array, the result is undefined. However, it is also legal for a pointer to point to `NULL`.

It is common to use the increment,`++`, or deincrement, `--`, operator on pointers for skipping through the values of an array sequentially:

```
...
p = &a[0];
for(i=0;i<sizeof(a)/sizeof(int);++i)
    *p++ = i * i;
assert(a[2] == 4);
```

Note, while there is no array bounds checking in C and the behaviour of addressing values beyond the limits of an array is strickly undefined, EiC attempts to be pointer safe (see section § 3.12.7, pg: 71).

The difference between two pointers, of the same class, will result in an integral value that represents the number of objects between the two address:

```
assert((p+1)-p == 1);
```

In EiC, as for ISO C, the difference is represented as the signed integral type `ptrdiff_t` defined in `stddef.h` (section § 4.1.10, pg: 117). It is illegal to add or multiple two pointers together. An integral value can be subtracted from or added to a pointer but it is illegal to subtract a pointer from an integral value:

```
...
int *p, *q;
p+20;     /* okay: results in the address of the 21st object*/
20+p;     /* okay: results in an address */
p - 10;   /* okay: results in an address */
p - q;    /* okay: results in an integer */
p * q;    /* error: incompatible types */
p+q;      /* error: incompatible types */
10 - p;   /* error: incompatible types */
```

Certain conversions are permitted. Pointers can be assigned to pointers of a different class, but without an explicit cast, EiC will issue a warning and a pointer can be cast explicitly to an integral value:

```
int *p; char * c;
```

```
...
p = c;  /* warning: Suspicious pointer conversion */
p = (int *)c;  /* okay */
printf("%p %ld",p, (long)p);
```

Any pointer can be compared with the integral value 0 and without the use of a cast:

```
#define NULL 0
...
if(p == NULL)
   ...
```

3.12.6 Void types

The concept of no value, or no argument, is expressed through the use of the `void` specifier. For example, a function that returns no value and receives no arguments would be declared as:

```
void f(void) { printf("Hello, world!\n");}
```

Also, in EiC, as for ISO C, there exists the `void` pointer. A `void` pointer is a generic pointer and any pointer to any object may be converted to a `void` pointer without a cast:

```
int *p; void * q;
q = p;  /* okay */
```

In EiC, pointers may be assigned to and from `void` pointers and may be compared to them without the use of an explicit cast.

3.12.7 Array types

An array is a sequences of objects of the same type. For example, an array of 10 floats is defined as:

```
float ar[10];
```

where the array begins with index zero and its elements are referenced via a primary expression: `ar[0]`, `ar[1]`, ..., `ar[9]` and the subscripts must be of an integral type. Further, arrays cannot be constructed from `void` types or functions.

EiC supports multidimensional arrays and there is no artificial limit on the number of dimensions that can be used nor on the physical size of an array. Array sizes are only limited by the amount of memory available. Multidimensional arrays are declared as arrays of arrays (which should not be confused with an array of pointers to arrays):

```
int a3d[3][5][10];
```

The array `a3d` contains 3 planes of 5 rows of 10 columns of integers. Its type is an array of 3 arrays of 5 arrays of 10 integers. Array elements are stored in a block of consecutive storage in row-major form and the last subscript varies the fastest.

It is useful to remember that array names are effectively treated as pointer constants except when used as an operand to the sizeof operator: `a3d` will evaluate to the address of the first element of the array – with type pointer to an array of 5 arrays of 10 integers; `a3d[i]` will evaluate to the address of the first element in `ith` plane – with type pointer to array of 10 integers; `a3d[i][j]` will evaluate to the address of the first element in the `jth` row of the `ith` plane – with type pointer to integer; and `a3d[i][j][k]` will evaluate to the `kth` element in the `jth` row of the `ith` plane – with type integer.

The `sizeof` operator when applied to an array returns the size of the entire array in bytes and not the size of a pointer:

```
assert(sizeof a3d == 3 * 5 * 10 * sizeof(int));
assert(sizeof a3d[0] == 5 * 10 * sizeof(int));
assert(sizeof a3d[0][0] == 10 * sizeof(int));
assert(sizeof a3d[0][0][0] == sizeof(int));
```

In general, to obtain the address of individual elements within an array the address operator `&` is used and `&a3d[i][j][k]` will evaluate to the address of the `kth` element in the `jth` row of the `ith` plane.

In EiC, `X[y]` is identical to `y[X]`. This is because the expression `X[y]` is identical to `*(X+y)`, which is identical to `*(y+X)`:

```
assert(a3d[2] == 2[a3d]);
```

Extending this notation to the next dimension shows that `X[y][z]` is identical to `*(*(X+y)+z)` and so forth.

Incomplete Arrays

In C, an incomplete array is an array whose size is not defined. It can only be referred to and only the first dimension may be missing. It size must be completed by a definition or by initialization:

```
extern int a[];   /* declaration */
int a[5];         /* definition */
int A[][2] = { {0,1}, {1,2}, {2,3} };
int B[]    = { 0,1,2,3,4,5,6,7,8,9};
```

Note, for 32-bit integers, `sizeof(A) = 24`, `sizeof(A)/sizeof(int) = 6`, `sizeof(B) = 40` and `sizeof(B)/sizeof(int) = 10`.

Array Bound Checking

EiC is pointer safe. This means EiC catches most array bound violations; for example:

```
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
```

EiC does this through inheritance. When arrays are allocated or memory is allocated by malloc etc, the size of the allocated piece of memory is known and retained. This information is passed along during assignments etc. For example, in the assignment `p = &a[5]`, `p` not only gets assigned the address of `a[5]`, it also inherits `a`'s range.

To detect array bound violations as efficiently as possible, EiC does not concern it self with the values held or produced by pointers, it worries about address values only when pointers 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
...
```

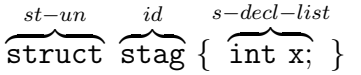
3.12.8 Structures and Unions

A structure is an instance of a sequence of named data types collected into a template – analogous to a Pascal record. A union is a data type that is similar to a structure but at most will contain only on member of its aggregation – it is handy for declaring a variable that may contain different types at different times. Structures and unions provide the way of extending the number of data types available.

Syntax:

st-un-spec:
    st-un [*id*] { *s-decl-list* }
    st-un id
st-un: `one of`
    `struct union`
s-decl-list:
    st-decl
    s-decl-list st-decl
st-decl:
    spec-qual-list spec-declor-list ;
spec-qual-list:
    type-spec [*spec-qual-list*]
    type-qual [*spec-qual-list*]
spec-declor-list:
    st-declor
    spec-declor-list , *st-declor*
st-declor:
    decl
    [*decl*] : *const-expr*

The underlined section in the above syntax, indicates that EiC does not support structure bit fields. The identifier, *id*, in the structure or union specifier *st-un-spec* is the tag name for the structure or union and each tag name must be unique. The scope of the tag will extend to the end of the block in which it is defined (see § 3.5, pg: 52). Note, tag names exist in a different name space from other variables. In ISO C, identifiers are grouped into at least four name spaces: 1) variables names, 2) structure, union and enumeration tag names, 3) labels for goto statements and 4) structures and unions have their own area for member names. See § 3.6, pg: 53 for further information.



The above definition declares a structure template, which contains just one member of type integer named `x`. Members can be any object type, including other structures or unions, but they can't be functions. A given member name may appear only once in any given structure or union. The above declaration also defines two variables `s1` and `s2` and the type specifier `struct stag`.

The type specifier `struct stag` can now be used to declare further variables:

```
struct stag a, b, *c;
```

where `a` and `b` are structure variables; and `c` is a pointer to a structure of type `struct stag`.

If the production *st-un id* is used without the preceding { *st-decl-list* }:

```
struct node;
```

an incomplete type is specified. A structure or union may not contain a member of incomplete type; that is an object of unknown size, but they can contain pointers to incomplete types; one advantage of this is in forward referencing – when for example, creating linked lists:

```
struct node {
    int a;
    struct node * next;
};
```

A structure or a union without a tag will form a unique type that can only be used in context of its declaration (however, see also the discussion below concerning structure and union compatibility):

```
struct {
    struct node *list;
    int count;
} head;
```

Although `typedef` names will be discussed in detail in § 3.12.9, pg: 78 they are of interest here because they provide a handy way to form structure or union type specifiers:

```
typedef struct node {
    int a;
    struct node * next;
} node;
```

The *typedef-name* `node` may appear anywhere the type specifier `struct node` can. Note, the first occurrence of the identifier node is entered into the tag-name space while the third is entered into the common variable name space. As these identifies exist in different name spaces they cause no conflicts (because context can be used to disambiguate their proper usage):

```
struct node a;
node b;
```

The above variables `a` and `b` are of the same type.

Structure and Union compatibility

Generally, each declaration for a structure or union type specifier creates a new type, which is not compatible with any other type specifier. For example, in ISO C, the following variables `x` and `y` are different:

```
struct {int x, y;} x;
struct {int x, y;} y;
```

This means that, in ISO C, the following is illegal:

```
struct {int x, y;} x;
struct {int x, y;} x;  // error: re declaration of variable ‘x’.
```

Because EiC is interactive, it needs to be more flexible than this, so it defines that two structures or unions to be compatible if they have the same type specifier or if they contain the same number of members of the same type, with the same names and in the same order. Hence, the re-declaration of variable `x` is not considered an error by EiC, and from the example before the last, the variables `x` and `y` are considered to be compatible. This definition is comparable to the ISO C definition for compatibility of structures or unions declared in separate source files.

Structure and Union assignment

Structures and unions can form modifiable lvalue expressions (if they have not been defined as constants or have members which have a `const` qualifier); that is, they can appear on the left-hand side of an assignment:

```
...
typedef struct { int a, b;} ab_t;
ab_t a1, b1;
```

```
const ab_t  a;
a = a1;    // error: Illegal assignment operation
a1 = b1;   // okay: copy the contents of b1 into a1, but b1 and a1
           // must be compatible
...
```

Initialization of Structures

The members of a structure can be initialized from the members of a compatible structure, `ab_t c = a1`, or from a brace inclosed list of constant expression initializers in order of the members: `ab_t c = {5,10}`. If there are fewer initializers than members, then the trailing members will be initialized to zero: `ab_t c = {5}`, is equivalent to `ab_t c = {5,0}`.

An array of structures can be initialized:

```
int f() {return 1;}
int i1 = 1,i2 = 2,i3 = 3;
struct {
    int *v;
    int (*p)();
}arg[3] = { {&i1,f}, {&i2,f}, {&i3,f}};
```

The inner sets of braces in this instance could have been dropped. However, the intention is clearer if they are retained.

Initialization of Unions

A union can be initialized from another compatible union or by a constant expression but the constant expression initializer must be brace-enclosed and be compatible with the first member of the union.

```
union {
    char a;
    int b;
    float c;
}un[3] = { {'a'}, {'b'}, {'c'}};
assert(un[0].a == 'a' && un[1].a == 'b' && un[2].a == 'c');
```

Note, that assigning to one element of a union makes all other elements have undefined values.

Structure and Union member access

The members of a structure or union are referred to by the selection operators `.` and `->`, which is a minus sign followed by >; for example:

```
struct {int a,b;} a, *b;
b = &a;    // address operation
a.a  = 5;
assert(b->a == a.a);
```

The operators `.` and `->` connects the structure name to a particular member and it is an error to reference a member of a structure or union that does not appear in the template of the structure or union.

Since `b` is a pointer to a structure the dereferenced type is a structure:

```
assert(b->a == (*b).a);
```

Structures and unions can be returned from functions:

```
EiC 1> typedef struct {int a;} a_t;
EiC 2> a_t f(void) { a_t a  = {5}; return a;}
EiC 3> f().a;
        5
EiC 4> f().a = 5;
Error in ::EiC:: near line 4: Illegal assignment operation
```

However, as can be seen from the previous example, they are not lvalues since they cannot appear on the left hand side of an assignment operator. Generally, `a.y` will form an lvalue if `a` is an lvalue and `y` is not an array identifier. A conditional operator, a comma operator or even an assignment operator can be used to produce a structure or union that is not an lvalue; for example:

```
EiC 1> int i = 2; struct {int a;} a = {15}, b = {69};

EiC 2> (i,b).a;  // comma operator
        69
EiC 3> (i,b).a = 6;
Error in ::EiC:: near line 3: Illegal assignment operation

EiC 4> (i == 2 ? a:b).a;  // conditional operator
        15
EiC 5> (i == 2 ? a:b).a = 5;
Error in ::EiC:: near line 5: Illegal assignment operation
```

```
EiC 6> (a=b).a;         // assignment operator
        69
EiC 7> (a=b).a = 5;
Error in ::EiC:: near line 7: Illegal assignment operation
```

Structures and Unions as parameters

Structures and unions are passed as arguments to functions by value:

```
EiC 1> #include <assert.h>
EiC 2> typedef struct {int a;} a_t;
EiC 3> int f(a_t a) { a.a++; return a.a;}
EiC 4> a_t a = {5};
EiC 5> assert(f(a) == a.a + 1);
```

However, it is generally more expedient, because a copy of the object does not have to be created, and more commonly practiced to pass structures and unions to functions by reference using a pointer:

```
EiC 6> :rm f  // remove f, before re declaration of new type
EiC 7> int f(a_t *a) { return a->a;}
EiC 8> assert(f(&a) == a.a);
```

Structure and Union layout

The members of a structure have addresses offset from the beginning of the structure that increase in order of declaration. The members of a union all begin at offset 0. Hence, the size of a union is equal to the size of its largest member. The address of a structure or a union coincides with the address of the first member:

```
EiC 1> #include <assert.h>
EiC 2> struct {char a; double x;} a;
EiC 3> char *p = (char *)&a;
EiC 4> assert(*p == a.a);
```

Often padding may appear between the members. This is know as the alignment problem; for example:

```
EiC 5> sizeof(a);
        12
EiC 6> sizeof(char) + sizeof(double);
        9
```

This occurs because on my system doubles must be aligned so that their address is a multiple of 4. On other systems this maybe 1, 8, etc. Padding can occur anywhere except at the beginning of a structure. In EiC, the padding occurs between consecutive members. The alignment of a union or a structure is equal to the maximum alignment of its members.

3.12.9 Typedef-name specifier

The *typedef-name* facility introduces synonyms for other type specifiers. It defines identifiers for types.

Syntax:

typedef-name:
    id

Declaring an identifier as a type allows that identifier to be used anywhere the original *type-specifier* could. However, it can't be mixed with other *type-specifiers*:

```
typedef float A[3]; // A is an array of 3 floats
typedef int  B[];   // B is an array of int
A *pa;              // pa is a pointer to an array of 3 floats
B *p[2];            // p is a 2-element array of pointers to
                    //   arrays of int of unspecified size
```

A *typedef-name* can be masked by the redefinition of its identifier at a higher scope level and on scope reentry the original *typedef-name* will again be visible:

```
typedef short S;
...
{
    char *S;   // new definition for S okay
    ...
}
```

It is illegal to mix a *typedef-name* with other type specifiers:

```
typedef char string[80];
unsigned string a;      // error: invalid type specification
```

Typedef-names can be used to specify the return type of a function:

```
EiC 1> typedef short (*S)(int x, int y);
EiC 2> S f(void);
EiC 3> :show f
f -> dec_Func (
        void
        ) returning * dec_Func (
                x: int ,
                y: int
                ) returning short
```

The above shows that `f` is a prototype declaration for a function that has a `void` parameter and returns a pointer to a function that returns a `short` integer and has two integer parameters named `x` and `y`.

3.13 Storage Class

Syntax:

store-class: `one of`
`auto extern register static typedef`

In EiC, as for ISO C, it is expected, but not compulsory, that the storage class of an object precedes other forms of *declaration specifiers*. Each object can have only one storage class specifier and with the exception of the storage class `typedef`, it determines an identifier's scope or extent(§ 3.5, pg: 52) and linkage.

auto: Automatic objects are local to the block that they are declared in. All objects declared within a block have by default automatic storage class unless otherwise specified. It is an error to declare a variable automatic outside a block – at the level of the function definition – and redundant to do so within a block.

```
auto int i; // error: Illegal storage class usage
{
    auto int i;   // okay
}
```

register: Automatic objects can be declared as a `register`, with the intention of being stored for fast access, but unlike the `auto` storage class, they can be used in parameters declarations. However, as EiC's virtual machine is stack based and not register based, declaring an automatic object to have storage class `register` – although allowed – has no real meaning. According to ISO standards it is illegal to take the address of a register object and thus EiC enforces this rule:

```
{
    register int i, *p;
    p = &i;       // error: cannot apply & to a register
}
```

extern: When used within a block, `extern` specifies that the storage for the object is specified elsewhere and if no external declaration is visible its linkage will be external. Otherwise, in ISO C, the object will have no linkage and be unique to the block it is declared in, but in EiC it will still be treated as an external object and which is not private to the block it is declared in. As an example of how the storage class `extern` can be used in both ISO C and EiC, consider the following program:

```
#include <assert.h>
int i = 5;
void T()
{   extern int i,j;   // Note, forward declaration of j
    i = 7;
    j = 10;
}
int j = 3;
int main(void)
{   assert(i == 5);
    assert(j == 3);
    T();
    assert(i == 7);
    assert(j == 10);
    return 0;
}
```

Also, as much as possible, EiC attempts to follow ISO recommendations and therefore the initialization of an extern variable is allowed:

```
extern int x = 5;   // okay
```

This forces the reference of `x` into a true definition. In EiC if no subsequent defining occurrence appears for an external variable, it to becomes the defining occurrence to that object.

static: Static objects may appear in declarations local or external to blocks. When used in the declaration of functions or global variables, `static` means that these identifiers will not be exported outside the current linkage unit or block in which they are declared. Hence, all global variables and functions declared static within an include file are private to that file, and are only visible within the scope of that file. Therefore, in general,

their names will not clash with the names of any global variable visible from the EiC interpreter. For example, consider the following program in `f1.c`:

```
#include <assert.h>
int p = 7;
#include f2.c
int main()
{
    setp(33);
    assert(getp() == 33);
    assert(p == 7);
    return 0;
}
```

where the contents of `f2.c` are:

```
/* private methods and data */
static int p;
/* public methods and data */
void setp(int x) { p = x;}
int getp() { return p;}
```

From the example just given, it is seen that the object `int p;` in file `f2.c` does not confict with the same object `int p;` previously defined in `f1.c`. This is because they are in different storage classes, see § 3.14, pg: 81 for further details on class conflicts.

With respect to automatic objects, defining a local object to be static also has meaning; as such objects retain their values across exit from and reentry to functions and blocks:

```
EiC> int i = 10;  // global variable
EiC> int f(void) { static int i = 1; return i++;}
EiC> while(i--) printf("%d",f());
```

results in the output: `12345678910`.

typedef: A typedef declaration, attributes a type to an identifier and thereafter, this *typedef-name* may appear anywhere the same type specifier may have appeared; see section § 3.12.9, pg: 78.

3.14 Default storage class and class conflicts

In ISO, if no storage class is specified for an object or function then the appropriate storage class will be determined from the current scope level. For all variables and functions

declared at level file, their default storage class is `extern`. Wthin a block the default class for objects is `auto`, for function prototypes it is `extern` and parameters, while having extent within the block belonging to a function, have no storage class.

Because in EiC the concept of linkage is different than that of a true compiler, there are no default storage specifiers. Within blocks, objects definition remain private unless specified `extern`, function declarations are exported to the outer most level and parameters are treated as specified for ISO C. Otherwise, all storage classes must be explicity stated.

The following table shows what EiC deems appropriate with respect to the storage class of the same object `int i` declared in two different files, and where file `f1.c` includes `f2.c` after the declaration of object `int i`:

```
// file f1.c
int i;
#include f2.c;
// end f1.c

// file f2.c
int i;
// end f2.c
```

A conflict between the same object or function difinitions declared in different files will generate an error, if it induces a condition where (1) a single object or function is owned by two files or (2) a public object or function, projects into the scope of a private object or function. Function prototypes (declarations) are handled more loosely and can be mutliply defined in different files. The following table is used to demonstrate the various conditions that can occur and equally applies to object and function definitions:

			f2.c	
		int i;	static int i;	extern int i;
f1.c	int i;	error	okay	okay
	static int i;	error	okay	error
	extern int i;	okay	okay	okay

In the third row of the above table, the object `int i` with no specified storage class is held constant in file `f1.c`, while it is varied for three different instances of file `f2.c`. When the object has no specified storage class in either file, it is consider an error because it is introducing a condition of a single object having two owners and this is not allowed. When it is declared in `f2.c` as `static`, there are no problems, because EiC sees two distinct variables, a private object owned by file `f2.c`, and a public object owned by file

`f1.c`. When the object in file `f2.c` is declared `extern`, this is also allowed, because file `f2.c` is explicity stating that it does not own the object – so there can be no conflict. The other rows are interpreted in similar ways. Note however, that when the object is declared `static` in file `f1.c` but has no declared storage class in file `f2.c` is also error. This is because the public object in file `f2.c` is visible within file `f1.c` and hence is inducing a situation in that there is two objects of the same type within the same scope, and this in not allowed either.

If `f2.c` had been included into `f1.c` before the declaration of the object `int i`, the roles of `f1.c` and `f2.c` in the above table would be have been reversed.

3.15 Type qualifiers

Syntax:

type-qual: `one of`
`const volatile`

Type qualifiers are used to specify extra information about a type and may appear with any type specifier.

const: The `const` qualifier is used to specify that a type is constant and therefore its value is not to be modified after initialization:

```
EiC 1> const int a = 5;
        (void)
EiC 2> a = 6;
Error in ::EiC:: near line 2: Illegal assignment operation
```

The variable qualified with `const` forms a non modifiable lvalue. With respect to simple assignments, the left operand cannot have a `const` qualifier. If the left and right operands are both pointers, the type pointed to by the left operand must be compatiable with the right operand and also have all the same qualifiers. Therefore, you can't assign `int const *` to `int *` without a cast:

```
EiC 3> const int *p;
        (void)
EiC 4> int * q;
        (void)
EiC 5> q = p;
Error in ::EiC:: near line 5: Assignment loses a const qualifier
```

Futher, when dealing with pointers, you have the opportunity to either declare the pointer constant, or what the pointer is pointing to as constant; for example:

```
EiC 1> int  a, * const p = &a;
EiC 2> const int *q = &a;
EiC 3> :show p
p -> const * int
EiC 4> :show q
q -> * const int
```

In the former case, `p` is a constant pointer to an interger. This means that `p` can't change in value, but the value of what it is pointing to can. In the latter case, `q` is a pointer to a constant integer. This means that `q`'s value can change but not the value it is pointing to. The following helps to highlight the various restrictions imposed by the `const` qualifier on `p` and `q`:

```
EiC 5> *p = 33;
        33
EiC 6> *q = 44;
Error in ::EiC:: near line 6: Illegal assignment operation
EiC 7> p = q;
Error in ::EiC:: near line 7: Illegal assignment operation
EiC 8> q = p;
        4a024
```

volatile: The `volatile` qualifier can appear with `const` and it is used to inform the compiler that the specified object may have it's value changed from external sources. In EiC, the `volatile` keyword is simply ignored.

3.16 Variable declaration placement

In EiC, you must declare all variables at the beginning of a block or at any position outside a function block.

Syntax:

{ *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

3.17 Function declarations

In this section I will first present a quick overview to function usage in EiC before looking at the details.

EiC attempts to be type safe; therefore, in order for EiC to check the validity of a function call during compilation, it is important that the function prototype be available. In EiC, the prototype form can be extracted from either the function definition or declaration. Hence, all function declarations must be in prototype form; that is, provide the name of the function, the types of its parameters and the return type of the function.

The two following prototypes are considered equivalent:

```
int swap(int *, int *);    /* prototype declaration */
int swap(int *a, int *b);  /* full prototype declaration */
```

The latter form is referred to as a declaration in full prototype form, because it includes the parameter names. However, for convenience and to increase backward compatibility, EiC, allows one type of non-prototype declaration:

```
void f();   /* allowable non-prototype declaration */
```

The above declaration will be compatible with other prototype declarations and definitions if: it matches the return type; the arguments declared in the prototype form will not be subject to typical argument conversions, such as `float` to `double`; and the prototype form does not declare a variadic function:

```
EiC 1> int round(double x) { if(x>0) return x+0.5; else return x-0.5;}
EiC 2> int sum(int x, int y) { return x + y;}
EiC 3> int foo(float var) { return var;}
EiC 4> int (*pf)();    // declare a function pointer
EiC 5> pf = round;
EiC 6> pf = sum;
EiC 7> pf = foo;
Warning: in ::EiC:: near line 7: Suspicious pointer conversion
```

EiC deems the assignment on line 7 to be suspicious, because the function `foo` accepts an argument of type `float`, and because all floating point values passed via `pf` will be automatically cast to type `double`. Also, as seen above, allowing a function declaration to take no variables is different from declaring it to take `void`. In the latter case, it implies that the function will accept no arguments; in the former case, it means that the function will accept any number of arguments, which do not undergo automatic conversion. The advantage of the former is obvious when working with a function pointer that may point to various other types of functions – although it does have limitations.

As already stated, before any function can be called from another function, EiC must have either processed the function's definition or the prototype form of the function. This is because there are no implicit parameters in EiC, EiC carries out strict variable type checking and it verifies all parameter types being passed between functions; for example:

```
#include <stdio.h>
void f() { g(); }    /* error: unknown identifier g */
void g() { printf("Hello, world!\n"); }
```

To fix this, two alternatives are possible: 1) place `g`'s definition before `f`'s, or 2) add `g`'s prototype either before `f`'s definition or within `f`:

```
#include <stdio.h>
void g(void);
void f() { g(); }    /* okay, g is now known */
void f2() {void g(); g();}  /* okay */
void g() { printf("Hello, world!\n"); }
```

Choice 1 is okay here, but would fail if the two functions were mutually recursive; therefore, choice 2 is the more general. However, in EiC there are no private declarations for functions, as function `f2` is implying. In EiC the above definition for `f2` is identical to:

```
void f2() {extern void g(); g();}  /* okay */
```

Hence, all function declarations within a function are exported to the level of a function definition.

3.18 Function types

In EiC, there are basically two types of functions: 1) interpreter functions, such as those supplied by the user and 2) builtin functions, which get linked into EiC at compile time. Naturally, builtin functions run a lot faster than interpreter functions. All builtin functions must be prototyped, via including the appropriate header file, before they are used, and as discussed with respect to the EiC `show` command on page 18.

Although, EiC attempts to make these two forms as invisible to the user as possible, EiC uses its own runtime stack for processing information and therefore, there is only one restriction on what can be passed to a function: you can't pass a structure or union by value to either a builtin function or to an interpreter function as part of the optional argument list in a variadic function call (see below).

3.19 Function definition

EiC has a more limited view of a function definition, *func-def*, and function declaration than specified by the ISO C standard. The ISO C grammar for a function definition is:

func-def:
decl-spec decl [*decl-list*] *comp-stmt*
decl [*decl-list*] *comp-stmt*

The parts of the above grammar that have been underlined are not included in EiC's grammar. This means that EiC does not recognize the old style of C function definition and that all function declaration must explicitly state their return type, which of course may be `void`. For example:

```
EiC> double sqr(double x) { return x*x;}
```

defines the function `sqr` to take a double for an argument and to return a double to its caller. Unlike C, all function definitions must explicitly specify their return type:

```
product(int x, int y)   /* error: implicit return type */
{
    return x * y;
}
```

The correct definition is:

```
int product(int x, int y)   /* explicit return type */
{
    return x * y;
}
```

Currently, C allows for the old C style and the new C style function definitions. With EiC and C++ the old style is not supported; that is, only the *parameter-type-list* is parsed:

```
int product(x, y)   /* error: old C style */
int x, y;
{
    return x * y;
}
```

The correct definition is:

```
int product(int x, int y)   /* New C style, parameter-type-list */
{
    return x * y;
}
```

3.20 Function parameter type list

The *parameter-type-list* that appears in function definitions and declarations has the following syntax:

parm-type-list:
    parm-list
    parm-list `, ...`
parm-list:
    parm-decl
    parm-list `,` *parm-decl*
parm-decl:
    decl-spec decl
    decl-spec [*abs-decl*]

The *parameter-type-list*, is a list of parameter declarations separated by commas. A parameter declaration must be in prototype syntax: declaring the type of the object and optionally its name. If the parameter list ends with a set of three ellipses, ..., then the function may accept any number of parameters of any type:

```
int printf(const char * fmt, ...);  /* variadic prototype */
```

However, note that for variadic function declarations and definitions, there must exist at least one named parameter, and in EiC it is illegal to pass a structure or union by value as part of the option argument list to a variadic function.

In EiC, as in ISO C, parameters are at the same scope level as the identifiers declared just after the beginning of the compound statement in the function definition. Therefore, it is illegal for a parameter name to be redeclared in the opening compound statement (but within inner blocks it is allowed):

```
int f(int n)
{
    int n;          /* error: re declaration of parameter ‘n’ */
    ...
    {
        int n;      /* okay here */
        ...
    }
}
```

If a parameter is declared to be an array of type `x`, it will automatically be cast to be a pointer to type `x`. If a parameter is declared to be a function returning type `x`, it will be cast to be a pointer to a function returning type `x`. The following are all legal parameter and function definitions:

```
int h(int (*x)(void)) { return (*x)();}  /* traditional */
int g(int (*f)(void)) { return f();}     /* hybrid style */
int f(int g2(void)) { return g2();}      /* modern style */
```

3.21 Function return type

Functions in EiC, like in C and C++, may return any type, a structure, a union, a pointer, etc. However, a function may not return an array, another function or an lvalue; that is, a function call cannot appear on the left side in an assignment expression

```
f() = x;        /* an illegal assignment expression */
```

The return type of a function is governed by its return statement.

```
#include <stdio.h>
typedef struct { int a,b; }ab_t;

ab_t * f(void) {
        static ab_t ab = {222,333};
        return &ab;
}
ab_t g(void) {
        static ab_t ab = {444,555};
        return ab;
}
int main(void) {
        printf("%d %d %d %d\n",
                f()->a,f()->b,
                g().a,g().b);
        return 0;
}
```

The return value, if possible, will be cast to agree with the return type, otherwise an error will be flagged. In C, a return statement with no expression causes control to be returned to the caller, but no useful value. In EiC, a function that does not return `void`, will return the last value on the stack, but without the explicit return statement, the result will most likely be garbage:

```
int s1(int x, int y)     { x + y;}  /* okay  */
int s2(float x, float y) { x + y;}  /* will return garbage */
int s3(float x, float y) { (int)(x + y);}  /* okay */
int s4(float x, float y) { return x + y;} /* correct */
```

Because the first three functions do not use an explicit return statement, EiC will issue a warning against their use.

The return type must be in agreement with the return type of the function. If a function has been declared to return type `void`, it is an error to attempt to return any value.

```
void f() { return 2;} /* error: illegal cast operation */
```

According to ISO C, the returning of a `void` statement must be tolerated:

```
void f() { return ;} /*okay, empty statement evaluates to void*/
```

It is legal in ISO C for a parameter and or the return type to be a declaration:

```
void sparm(struct s { int a, b;} ab)
{
    printf(" a = %d, b = %d\n", ab.a, ab.b);
}

struct {int a, b;} srtn()
{
    static struct {int a, b;} x;
    return x;
}
```

Although the above functions are legal, it's best to avoid writing such obscure code.

3.21.1 Function flow-of-control analysis

After EiC has compiled a function into bytecodes, it performs a flow-of-control analysis to check that control does not reach the end of non-void functions – as well as other checks, such as looking for unreachable code:

```
int f(int a)
{
    switch(a) {
      case 1: return 1;
              a = 5;        /* warning: unreachable code */
      case 2: return 2;
      case 3: return 3;
    }
} /* warning: control reaches end of non-void function ‘f’*/
```

In the example given above, if the `switch` statement (see § 3.24.3, pg: 94) had a `default` label, which returned control to the caller then control would not have been detected to reach the end of the function.

3.22 Type names

Syntax:

type-name:
    spec-qual-list [*abs-decl*]
abs-decl:
    pointer
    [*pointer*] *dir-abs-decl*
dir-abs-decl:
    `(` *abs-decl* `)`
    [*dir-abs-decl*]`[` [*const-expr*] `]`
    [*dir-abs-decl*] `(` [*par-type-list*] `)`

A type name is a declaration without an identifier. It represents an abstract data type. `Type names` are used to specify various types. For example, in a unary expression, (§ 3.24.6, pg: 99), when `1)` type casting an object or function into another object or function:

```
EiC 1> -1;
        -1
EiC 2> (unsigned)-1;
        4294967295
```

or `2)` as the operand to the `sizeof` operator:

```
EiC 3> sizeof(struct {int x, y;});
        8
```

The following lists various example of typenames:

short int	a short integer
double *	a pointer to a double
float [5]	an array of 5 floats
int ()	a function returning int
int (*)()	a pointer to a function returning int
int (char *)	a function taking a char pointer and returning an int

`Type names` are also used by the EiC interpreter for displaying various identifiers associate with a given type, see the EiC `variables` command page 23. `Type names` are always enclosed in parentheses except when used as an operand to the EiC `variables` operator.

3.23 The address specifier operator `@`

In EiC, the address of where a variable is located can be specified via the address operator `@`:

```
float f @ dddd;
```

Defines `f` to be a variable of type `float` and which is stored at memory location `dddd`, where `dddd` must be an integral constant. If `dddd` is not a valid address within the scope of EiC, then `f` is undefined.

The constant address `dddd` is not simply an address conjured by the user. 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 the function must take void arguments and return void:

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

Defines `foo` to be a builtin function located at address `dddd`. For further examples see discussions on embedding EiC § 1.2.8, pg: 13.

3.24 Statements

EiC supports all the usual C statements.

Syntax:

stmt:
comp-stmt
label-stmt
select-stmt
iter-stmt
jump-stmt
expr-stmt

3.24.1 Compound-statement

A compound statement is used to form a block of code that has a different scope level, see § 3.5, pg: 52, than its environment and it has the form:

comp-stmt:
{ [*decl-list*] [*stmt-list*]}
stmt-list:
[*stmt*]+

Any variable declared in a compound statement is local to that block. If an identifier declared outside the block is the same as one declared in the block, the inner declaration will mask the outer. All identifiers defined in a block must be unique. Initialization of local variables must be explicit and will be performed each time the block is entered. Initialization of local `static` variables occurs once and at compile time, see storage class § 3.13, pg: 79. A compound statement can appear anywhere a normal C statement can and it does not require a terminating semi-colon.

3.24.2 Label Statement

The label statement is used to mark a position in the code where control can jump to; either by a `goto` or a `switch` statement.

Syntax:

label-stmt:
id `:` *stmt*
`case` *const-expr* `:` *stmt*
`default :` *stmt*

The *id* label is used to specify the target for the goto jump statement, see: § 3.24.5, pg: 96, and are used exclusively within the scope of a function. Labels also have their own name space, so label *id* values can only potentially conflict with other label values. No two labels within the same scope can be equal.

The `case` and the `default` statements are used exclusively within the `switch` statement, see section § 3.24.3, pg: 94.

3.24.3 Selection Statements

Program flow can be altered by using one of the selection statements:

select-stmt:
`if (` *expr* `)` *stmt*
`if (` *expr* `)` *stmt* `else` *stmt*
`switch (` *expr* `)` *stmt*

If-else statement

The *expr* in the `if` statement must be of arithmetic or pointer type. If statements are normally used to form two way decision statements:

`if (` *expr* `)` $stmt_1$
[`else` $stmt_2$]

If the value of *expr* evaluates to be non zero then program control will pass to $stmt_1$, else if the optional `else` statement is present and *expr* evaluates to zero then control will pass to $stmt_2$. Otherwise, control will pass to the first statement beyond the `if` statement. `If-else` statements are often nested so as to form a multi-way branch statement:

```
if(x  == 1) printf("A");
else if(x == 2) printf("B");
else printf("C");
```

Switch Statement

The `switch` statement is C's formal multi way decision statement:

```
switch( expr ) {
   case const-expr1: stmt1;
   [ case const-expr2: stmt2; ]
   [ default: stmt3; ]
}
```

It evaluates the *expr*, which must have integral type, and it compares the value against each of the `case` *const-expr*, which are constant integral expressions that get `cast` to the same type as *expr*. If a match is found, control is passed to that branch in the body of the `switch` statement. If no match is found, and there exists a `default` statement, then control is passed to it. Otherwise, none of the statements in the body of the `switch` will be evaluated.

No two `case` *const-expr* can be the same. The body of the `switch` statement, as shown above, is usually a compound statement, but it maybe just a single statement. Note also, the `default` statement and the `case` *const-expr* can occur in any order and that there is no limit on the number of `case` *const-expr* that can be used.

One feature of C's is that switches don't break automatically before each case *const-expr*. The `break` or `return` statements are therefore, often used to terminate execution of a `switch` statement. After termination, by either a `break` statement or completion of the last statement in the body of the `switch`, control is passed to the next C statement beyond the `switch`.

```
switch(x) {
   default: printf("X");
   case  1: printf("A");
            break;
   case  2: printf("B");
   case  3: printf("C");
```

```
      case  4: printf("D");
      case  5: printf("E");
   }
   printf("F");
```

A value of `x` equal to 1, entered above, causes `AF` to be printed. A value of 3 causes `CDEF` to be printed, and any value not in the set `[1,2,3,4,5]`, causes `XAF` to be printed.

3.24.4 Iteration Statements

C provides just three basic kinds of loops:

iter-statement:
`while (` *expr* `)` *stmt*
`do` *stmt* `while (` *expr* `) ;`
`for (` [*expr*$_1$] `;` [*expr*$_2$] `;` [*expr*$_3$]`;)` *stmt*

The conditional *expr* in each iteration statement must be of arithmetic or pointer type. With respect to the `for` loop, the conditional statement is *expr*$_2$, which is also optional.

While Statement

The purpose of the `while` statement is to repeatedly execute a statement until the conditional *expression* evaluates to zero:

```
int x = 10;
while( x ) {
   printf("x = %d\n",x);
   x = x - 1;
}
```

Do . . . While Statement

In the `while` loop, the conditional *expression* is evaluated before each iteration but in the `do...while` loop it is evaluated after each loop

```
int x = 10;
do {
   printf("x = %d\n",x);
   x = x - 1;
} while ( x );
```

For Statement

The most ubiquitous loop in C would have to be the `for` loop:

`for (` [$expr_1$] `;` [$expr_2$] `;` [$expr_3$]`;)` *stmt*

Where the expressions separated by semicolons can be loosely defined as:

$expr_1$: is usually an assignment expression.

$expr_2$: is a conditional expression.

$expr_3$: usually some form of update or modification rule.

For example:

```
for(x = 10; x > 0; x = x - 1)
     printf("x = %d\n",x);
```

Which is equivalent to the above `while` statement; that is:

$expr_1$`;`
`while (` *$expr_2$* `) {`
    stmt`;`
    $expr_3$`;`
`}`

When the conditional expression is left out of the `for` loop it will iterate forever or until control is transfered out of the loop by one of several forms of *jump* statements.

3.24.5 Jump Statements

jump-stmt:
    `goto` *id* `;`
    `continue ;`
    `break ;`
    `return` [*expr*] `;`

The `goto` jump statement is used to redirect program flow to the target identifier, which must be a label, see: § 3.24.2, pg: 93. The `continue` and the `break` statements may appear only in an *iteration* statement. While the `return` statement can appear anywhere within the body of a function.

Continue Statement

In an iteration loop the `continue` statement forces the start of the next iteration or *loop-continuation* of the inner most `while`, `do...while` or `for` loop. In a `for` loop, the next iteration resumes only after evaluation of the current iteration $expr_3$:

```
for(x = 10; x > 0; x = x - 1) {
     if(x > 5)
        continue;
     printf("x = %d\n",x);
}
```

Only allows the numbers 5,4,3,2 and 1 to be printed out.

Break Statement

The `break` statement causes termination of the innermost `while`, `do...while` or `for` loop and it passes control to the statement immediately following the `while`, `do...while` or `for` loop, see also § 3.24.3, pg: 94.

Return Statement

The `return` statement is used to return control from the current function back to the calling function. If an expression follows the `return` its value is returned also to the caller. When control reaches the end of a function, which has no `return`, it forces a return to the caller, see also § 3.21, pg: 89.

3.24.6 Expression Statement

Most C statements form an expression, and C has a particularly rich set of expression `operators`, which are the symbols that are used to represent operations. A missing expression is called a `null` statement, and has type `void`. In this section, the C operators are discussed along with their `precedence`.

Precedence, Associativity and nomenclature

In C, each `operator` has a precedence level, a rule of associativity; that is, order of evaluation and `operand` count. Precedence refers to the priority used to decide on how to associate operands with operators, while associativity refers to the order of evaluation of a succession of operators of a given type; that is, either from left-to-right or from right-to-left.

To summarise, the C operators are listed in decreasing order of precedence (where LR and RL are used to designate left-to-right or from right-to-left associativity):

Token	Associates	Expression type
identifiers, literals	na	primary
f(...) a[k] -> . ++ --	LR	postfix
! ~ ++ -- - + * & sizeof	RL	unary
(type-name)	RL	cast
* / %	LR	multication
+ -	LR	addition
<< >>	LR	shift
< <= > >=	LR	relational
== !=	LR	equality
&	LR	AND
^	LR	XOR
|	LR	inclusive OR
&&	LR	logical AND
||	LR	logical OR
? :	RL	Conditional
= += -= *= /= %= &= ^= |= <<= >>=	RL	Assignment
,	LR	Comma

Below is a brief classification of the operators supported by EiC.

Primary expressions

Syntax:

primary-expr:
id
constant
string
`(`*expr* `)`

Identifiers, numeric constants, string literals, and parenthesised expressions are all primary expressions. The various constant types and string literals are discussed in section § 3.9, pg: 55. The value of an identifier *id* is determined from its declaration as explained in section § 3.11, pg: 60. If the identifiers type represents an object, as opposed to a function, it will form an lvalue. If its type is not qualified with `const` (§ 3.24.6, pg: 105) and in the case of a structure or union it does not contain any members qualified with `const`, then the lvalue will be modifiable. A parenthesised expression consists of any expression surrounded by left and right parenthesis.

Postfix expressions

The *postfix expressions* group from left-to-right and they are typically used to form function calls, subscripting and for structure and union member selection:

postfix-expr:
primary-expr
postfix-expr `[` *expr* `]`
postfix-expr `(` [*arg-expr-list*] `)`
postfix-expr `.` *id*
postfix-expr `->` *id*
postfix-expr `++`
postfix-expr `--`

`()` funcname`()` Function call.

`[]` The array operator. See section § 3.12.7, pg: 69. The value in the bracket is used as an index into the array.

`.` The struct or union address operator, see § 3.12.8, pg: 72.

`->` The struct or union indirect selection operator. see § 3.12.8, pg: 72.

`++` The postfix increment operator, `y = x++`, will assign `x` to `y` and then increment `x` by one. `x` must be an lvalue.

`--` The postfix deincrement operator, `y = x--`, will assign `x` to `y` and then deincrement `x` by one. `x` must be an lvalue.

Unary expressions

The *unary-expressions* group right-to-left.

unary-expr:
postfix-expr
[`++`,`--`] *unary-expr*
[`&`, `*`, `+`, `-`, `~`, `!`] *cast-expr*
`sizeof` [`(` *type-name* `)`, *unary-expr*]

`*` The indirection operator yields the type of its operand, which must be a pointer. The resulting type is either an object or a function designator depending upon the pointer type.

`&` The address operator yields the address of its operand, which must be an `lvalue` or a function name. The result is a pointer to an object or a function. When the operand is an array of type T the result is a pointer to an array of T. The address operator is the inverse of `*`; that is, `x= *&x`.

`+` The unary plus operator serves no real purpose other than complementing the negation operator `-`. However, its operand must be of arithmetic type.

- The negation operator reverses the sign of its operand, which be of arithmetic type. The resulting type is also arithmetic. The negative of an unsigned value x results in an unsigned value, which is computed by subtracting x from its largest promoted value and adding one.

! The logical not operator, yields the logical negation of its operand. If the operand is zero, it results in one and vise-a-versa. The operand can have arithmetic or pointer type but the result will be of integer type.

~ The one's complement operator, whose operand must be of integral type. The bits of the operand are complemented; that is, every one bit becomes zero and every zero bit becomes one. If the operand is signed, the operator complements the bits after promoting the operand to its unsigned type.

++ The unary increment operator results in a value which is one greater than its operand, which must be an `lvalue` and leaves the operand incremented also. The operand maybe any arithmetic or pointer type. The result will be same as its operand but it will not be an `lvalue`.

-- The unary deincrement operator is the same as the unary increment operator expect the result and the operand is deincremented by 1.

sizeof() The `sizeof` operator returns the size in bytes of its operand, which must be a *type-name* (§ 3.22, pg: 91) surrounded by parentheses: `sizeof(int)`; or a *unary-expression*: `sizeof a`, where `a` is a variable.

The return type is an the unsigned integral constant `size_t` as defined in `<stddef.h>` (§ 4.1.10, pg: 117). The `sizeof(char)` is always 1, while the `sizeof` a structure or union results in the size of the structure or union in terms of bytes (see Structure and Union Layout, page 77).

Cast expression

A cast expression is either a *unary expression* or a *type-name* enclosed in parentheses followed by a *cast expression*.

Syntax:

cast-expr:
unary-expr
`(` *type-name* `)` *cast-expr*

Cast expressions can be used for type conversions. They are used to cast on type into another. That is, the following *cast-expression* operand is converted to the specified `name-type`, see also § 3.22, pg: 91.

Multiplication expressions

The multiplication operators group left-to-right.

mult-expr:
cast-expr
mult-expr `*` *cast-expr*
mult-expr `/` *cast-expr*
mult-expr `%` *cast-expr*

`*` The multiplication operator yields the product of two adjacent operands. The operands must have arithmetic type.

/ The division operator yields the quotient of the left operand divided by the right. The operands must have arithmetic type.

`%` The modulo operator yields the remainder of the left operand divided by the right. The operands must have integral type.

If the right hand operand for either the division or the modulo operator is zero then the result is undefined.

Additive expressions

The additive operators group left-to-right.

add-expr:
mult-expr
add-expr `+` *mult-expr*
add-expr `-` *mult-expr*

\+ The algebraic addition of two adjacent operands, yielding a sum. If one operator is a pointer then the other must have integral type and the result will be an address which is offset from the address operand by the number of objects determined from the non address operand. Otherwise, both operands must have arithmetic type.

\- The algebraic difference, which subtracts the right operand from the left. If the left operand is an address operator and if the right operand is another address operator, which must be of the same type, then the result is a `ptrdiff_t` as defined in `<stddef.h>` (§ 4.1.10, pg: 117) and will represent the number of objects between them – the result will be undefined if they do not point within the same array. If the right operand is an integral value then the same rules apply as for addition. Otherwise, both operands must have arithmetic type.

See also section § 3.12.5, pg: 67.

Shift expressions

The shift operators group left-to-right, and they yield the left operand arithmetically shifted left or right by the number of bit positions determined from the right operand. Both operand must have integral type:

shift-expr:
add-expr
shift-expr `<<` *add-expr*
shift-expr `>>` *add-expr*

<< The shift left operator. With each shift, a zero is inserted at the lowest bit that has been displaced.

>> The right shift operator. With each shift operation, if the left operand is an unsigned number, then zeros bits as shifted into the highest bits, else if the left operand is signed the sign bit remains the same – this may vary depending upon the implementation used to build EiC. Excess bits shifted too far are conceded to fall off the end.

Relational expressions

The relational operators group left-to-right. They perform a numeric comparison of two operands yielding either 1 or 0. The operands can be both of arithmetic type or of pointer type.

rel-expr:
shift-expr
rel-expr `<` *shift-expr*
rel-expr `>` *shift-expr*
rel-expr `<=` *shift-expr*
rel-expr `>=` *shift-expr*

Equality expressions

The equality operators group left-to-right. The same rules apply as for the *relational expressions*, with the addition that address operators can be compared with the value zero, or to a `void` pointer.

equal-expr:
rel-expr
equal-expr `==` *rel-expr*
equal-expr `!=` *rel-expr*

== Yields 1 if the left operand is equal to the right otherwise 0.

!= Yields 1 if the left operand does not equal the right otherwise 0.

Bitwise expressions

The bitwise operators group left-to-right. They perform logical operations on the bits of their operands.

inc-or-expr:
    xor-expr
    inc-or-expr `|` *xor-expr*
xor-expr:
    and-expr
    xor-expr `^` *and-expr*
and-expr:
    equal-expr
    and-expr `&` *equal-expr*

`|` Bitwise inclusive OR.
`^` Bitwise exclusive OR.
`&` Bitwise AND.

Logical expressions

The logical operators group from left-to-right. The logical operators test for ones and zeros and generates either a 1 or a 0. In both cases, short circuit logic is used; that is, the second operand of the logical operators is evaluated only if necessary.

log-or-expr:
    log-and-expr
    log-or-expr `||` *log-and-expr*
log-and-expr:
    inc-or-expr
    log-and-expr `&&` *inc-or-expr*

`||` Logical OR. If either operand yields 1, it yields 1, otherwise it evaluates to 0. A sequence of logical-or expressions will be evaluated from left-to-right until the first one yields 1. The remaining expressions are guaranteed not to be evaluated.
`&&` Logical AND. If either operand yields 0, it yields 0, otherwise it evaluates to 1. A sequence of logical-and expressions will be evaluated from left-to-right until the first one yields zero. The remaining expressions are guaranteed not to be evaluated.

Conditional expressions

Syntax:

cond-expr:
log-or-expr
log-or-expr `?` *expr* `:` *cond-expr*

The conditional operator is a type of `if...else` statement, see § 3.24.3, pg: 93, that can be used to form an `rvalue`.

```
if(a == 5)
    printf("a = 5\n");
else
    printf("a != 5\n");
```

Is equivalent to:

```
a == 5 ? printf("a = 5\n") :  printf("a != 5\n");
```

However, unlike a selection statement, the result can form an `rvalue`:

```
x = (a==5) ? 20 : 30;
```

With the conditional expression, if the value of the left most operand evaluates to none zero then the second operand is evaluated else the third operand is. The result of this `ternary` operator depends on the types of the second and third operands and will be cast to a common type. The result will inherit the qualifiers from both the second and third operands. If both arms are arithmetic the result is arithmetic. If they are structures or unions of compatible types the result is a structure or union of that type. If they are pointers they must be compatible and the result is a pointer. If one is a pointer and the other is zero then the result will be a compatible pointer. If one operand is a `void *` then the other must be a pointer or zero and the result will be a `void *`.

Assignment expressions

assignment-op:
`=`
[`*`, `/`,`%`, `+`, `-`, >>, <<, `&`,`^`,`|`] `=`

The assignment expression groups from right-to-left and never forms an `lvalue`. All but one of the assignment operators have the form `var op= exp`, where `op` is a compound assignment operator. The resulting type is always same as the left operand. The left operand must be a modifiable `lvalue` and will be evaluated only once. Also, any number of assignment operators may appear in an expression:

```
a = b = c = d;
```

=	Assign the value of the right operand to the left operand. Both operands can be arithmetic or both can be structures or unions of the same type. It is illegal to assign a value of a pointer to `const X` to an object of type pointer to `X` without an explicit cast.
*=	Assigns the product of the right and left operands to the left operand.
/=	Assigns the division of the left operand by the right to the left operand.
%	Assigns the remainder of the division of the left operand by the right to the left operand.
+=	Assigns the sum of right and left operands to the left.
-=	Assigns the difference between the left and right operand to the left operand..
<< =	Assigns the result of shifting the left operand left the number of bits specified by the right operand to the left operand.
>> =	Assigns the result of shifting the left operand right the number of bits specified by the right operand to the left operand.
&=	Assigns the bitwise `and` of the left and right operands to the left operand.
^ =	Assigns the bitwise exclusive `or` of the left and right operands to the left operand.
|=	Assigns the bitwise `or` of the left and right operands to the left operand.

Constant expressions

A constant expression must be able to be evaluated at compile time.

Syntax:

const-expr:
    cond-expr

Constant expression are required to form constant initializer expressions, enumeration constants, array bounds and for case labels. A constant expression may not contain an assignment, increment, decrement, function call or a comma expressions unless contained as the operand to the `sizeof` operator.

If the expression is to be `integral` its operands must be of integral type or a enumeration constant. Floating point constants can only appear if they are explicitly cast to an integral type or as an operand to the `sizeof` operator.

The address constant expression used in intializations can be formed from the null pointer; from the address of a static or external object or function; or via casts.

Chapter 4

Library support

4.1 Standard C libraries

This section describes how the standard C library is supported by EiC. Note, any function, macro or type that is underlined is currently not supported by EiC.

4.1.1 assert.h

The header files `<assert.h>` defines the following macro:

assert: `Synopsis`:

```
#include <assert.h>
void assert(int expression);
```

If `expression` is false, a message is printed to `stderr` and abort is called to terminate execution. However, EiC does not call abort. The source file and line number in the output message comes from the preprocessor macros `__FILE__` and `__LINE__`. If `NDEBUG` is defined when `<assert.h>` is included, the assert macro is ignored.

4.1.2 ctype.h

isdigit: `Synopsis`:

```
#include <ctype.h>
int isdigit(int c);
```

Returns 1 if `c` is in the set `'0' - '9'`. Otherwise it returns 0.

isupper: `Synopsis`:

```
#include <ctype.h>
int isupper(int c);
```

Returns 1 if `c` is in the set `'A' - 'Z'`. Otherwise return it returns 0.

islower: Synopsis:

```
#include <ctype.h>
int islower(int c);
```

Returns 1 if `c` is in the set `'a' - 'z'`. Otherwise returns 0.

isalpha: Synopsis:

```
#include <ctype.h>
int isalpha(int c);
```

Returns a 1 if `c` is in the set `'A' - 'Z'` or in `'a' - 'z'`. Otherwise returns 0.

isprint: Synopsis:

```
#include <ctype.h>
int isprint(int c);
```

Returns 1 if `c` is a printable character. Otherwise returns 0.

isalnum: Synopsis:

```
#include <ctype.h>
int isalnum(int c);
```

Returns 1 if `c` is in one of the sets `'a' - 'z'`, `'A' - 'Z'`, or `'0' - '9'`. Otherwise returns 0.

isspace: Synopsis:

```
#include <ctype.h>
int isspace(int c);
```

Returns 1 if `c` is in the set `' '`, `'\t'`, `'\n'`, `'\v'`, `'\f'`, `or '\r'`. Otherwise returns 0.

toupper: Synopsis:

```
#include <ctype.h>
int toupper(int c);
```

If `c` is a lower case alphabetic character, `touppper` returns its upper case equivalent, otherwise it returns `c`.

tolower: Synopsis:

```
#include <ctype.h>
int tolower(int c);
```

If `c` is an upper case alphabetic character, `tolower` returns its lower case equivalent, otherwise it returns `c`.

4.1.3 errno.h

The header `<errno.h>` contains manifest constants for error codes. The variable `errno` is a modifiable *lvalue* that has type `int`. It is set to zero on EiC startup, and it is used to report various runtime errors; for example:

```
#include <math.h>
#include <errno.h>
#include <assert.h>
int main(void)
{
    assert(errno == 0);
    sqrt(-3);
    assert(errno == EDOM);
}
```

The `<errno.h>` files declares the following macros and object.

EDOM: Which stands for domain error. It occurs if an argument is outside the domain of a function; for example: `sqrt(-2)`.

ERANGE: Which stands for range error and is used for reporting various numerical overflow and underflow errors. It occurs if the result of a `math.h` function (see § 4.1.6, pg: 111) cannot be represented as a double.

errno: Is a modifiable *lvalue* with type `int` that may or may not be a real identifiable object.

4.1.4 float.h

In C, floating point values are represented in the normalised form:

$$x = s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k}, e_{\min} \leq e \leq e_{\max}$$

where,

- s sign (± 1)
- b is the base or radix, typically 2, 8, 10 or 16.
- e the exponent, a value between a minimum $e_{\min}$ and a maximum $e_{\max}$.
- p precision (the number of base-b digits in the significand).
- f_k the significant digits.

The header file `<float.h>` declares the following macros:

FLT_RADIX: radix of exponent representing b.

FLT_ROUNDS:	Indicates the rounding mode for floats point values: -1, undetermined 0 toward zero 1 to the nearest 2 towards positive infinity 3 towards negative infinity
FLT_DIG:	number of digits of precision in a `float`
FLT_EPSILON:	smallest number x such that 1.0 + x != 1.0
FLT_MANT_DIG:	the number of *base-b* digits in the floating-point significand.
FLT_MAX:	maximum floating-point number
FLT_MAX_10_EXP:	maximum x such that 10^x is representable
FLT_MAX_EXP:	The maximum n such that `FLT_RADIX`$^n - 1$ is representable
FLT_MIN:	minimum normalised floating-point number
FLT_MIN_10_EXP:	minimum n such that 10^n is a normalised number.
FLT_MIN_EXP:	minimum n such that b^{n-1} is a normalised number.
DBL_DIG:	number of digits of precision in a `double`
DBL_EPSILON:	smallest number x such that 1.0 + x != 1.0
DBL_MANT_DIG:	the number of *base-b* digits in the floating-point significand.
DBL_MAX:	maximum double floating-point number
DBL_MAX_10_EXP:	maximum x such that 10^x is representable
DBL_MAX_EXP:	The maximum n such that `FLT_RADIX`$^n - 1$ is representable
DBL_MIN:	minimum normalised double floating-point number
DBL_MIN_10_EXP:	minimum n such that 10^n is a normalised number.
DBL_MIN_EXP:	minimum n such that b^{n-1} is a normalised number.

4.1.5 limits.h

The header file `<limits.h>` declares the following macros:

CHAR_BIT:	number of bits in a char
CHAR_MAX:	maximum value of char
CHAR_MIN:	minimum value of char
INT_MAX:	maximum value of int
INT_MIN:	minimum value of int
LONG_MAX:	maximum value of long
LONG_MIN:	minimum value of long
SCHAR_MAX:	maximum value of signed char
SCHAR_MIN:	minimum value of signed char
SHRT_MAX:	maximum value of short
SHRT_MIN:	minimum value of short

UCHAR_MAX: maximum value of unsigned char
UCHAR_MIN: minimum value of unsigned char
UINT_MAX: maximum value of unsigned int
ULONG_MAX: maximum value of unsigned long
USHRT_MAX: maximum value of unsigned short

4.1.6 math.h

The header `<math.h>` declares the following macro:

HUGE_VAL: A positive `double`. The largest representable floating point value. Not necessarily representable by a `float`.

The following functions are defined `<math.h>`

atan: `Synopsis`:

```
#include <math.h>
double atan(double x);
```

Returns the arc tangent of x.

ceil: `Synopsis`:

```
#include <math.h>
double ceil(double x);
```

Returns the smallest integer greater than or equal to x. The returned value is a double.

cos: `Synopsis`:

```
#include <math.h>
double cos(double x);
```

Returns the cosine of x.

cosh: `Synopsis`:

```
#include <math.h>
double cosh(double x);
```

Returns the hyperbolic cosine of x.

exp: `Synopsis`:

```
#include <math.h>
double exp(double x);
```

Returns the exponential function of x.

fabs: Synopsis:

```
#include <math.h>
double fabs(double x);
```

Returns the absolute value of x. The value returned is a double.

floor: Synopsis:

```
#include <math.h>
double floor(double x);
```

Returns the largest integer less than or equal to x. The returned value is a double.

sin: Synopsis:

```
#include <math.h>
double sin(double x);
```

Returns the sine of x.

sinh: Synopsis:

```
#include <math.h>
double sinh(double x);
```

Returns the hyperbolic sine of x.

sqrt: Synopsis:

```
#include <math.h>
double sqrt(double x);
```

Returns the square root of x.

tan: Synopsis:

```
#include <math.h>
double tan(double x);
```

Returns the tangent of x.

tanh: Synopsis:

```
#include <math.h>
double tanh(double x);
```

Returns the hyperbolic tangent of x.

log: Synopsis:

```
#include <math.h>
double log(double x);
```

Returns the natural logarithm of x.

log10: Synopsis:

```
#include <math.h>
double log10(double x);
```

Returns the base 10 logarithm of x.

pow: Synopsis:

```
#include <math.h>
double pow(double x, double y);
```

Returns x raised to the yth power.

atan2: Synopsis:

```
#include <math.h>
double atan2(double x, double y);
```

Returns the arc tangent of `y/x`.

4.1.7 setjmp.h

The header `<setjmp.h>` declares the following macro and type:

jmp_buf: An array of type suitable for holding the information needed to restore a calling environment.

setjmp: Synopsis:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

The `setjmp` macro save the calling environment in `env`. Zero returned from direct call; non-zero from subsequent call of `longjmp`.

The following macro function is defined in `<setjmp.h>`:

longjmp: Synopsis:

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Restore state saved by most recent call to `setjmp` using information saved in `env`. Execution resumes as if `setjmp` just executed and returned non-zero value `val`. Also, `longjmp(env,0)` is equivalent to `longjmp(env,1)`. If the function containing `setjmp` has terminated before the `longjmp` call is made then EiC's behaviour will be undefined.

An example usage of `setjmp` and `longjmp` is:

```
#include <stdio.h>
#include <setjmp.h>

jmp_buf env;

void dojump() {longjmp(env,1);}

void dosetjmp()
{
    switch(setjmp(env)) {
       case 0: printf("setjmp return 0\n"); break;
       case 1: printf("setjmp return 1\n"); return;
       default: printf("error\n"); return;
    }
    dojump();
}

int main()
{
    dosetjmp();
    printf("exit main\n");
    return 0;
}
```

Which should output:

```
setjmp return 0
setjmp return 1
exit main
```

4.1.8 signal.h

The header `<signal.h>` declares the following macros:

SIGABRT:	abnormal termination
SIGFPE:	arithmetic error
SIGILL:	illegal function image
SIGINT:	interactive attention
SIGSEGV:	illegal storage access
SIGTERM:	termination request sent to program

The header `<signal.h>` defines the following macro functions, which can be used to specify the action for the signal:

SIG_DFL: specifies the default action for the particular signal.

SIG_IGN: specifies that the signal should be ignored.

If a signal cannot honored its call, it returns `SIG_ERR`.

SIG_ERR: This macro is used as a return value to indicate an error.

EiC also supports POSIX.1 signals (see § 4.2.5, pg: 142).
The following functions are defined in `<signal.h>`:

signal: Synopsis:

```
#include <signal.h>
void (*signal(int sig, void (*handler)(int)))(int);
```

Install `handler` for subsequent signal `sig`. If `handler` is `SIG_DFL`, implementation-defined default behaviour is used; if `handler` is `SIG_IGN`, signal is ignored; otherwise function pointed to by handler is called with argument `sig`. signal returns the previous handler or `SIG_ERR` on error. When signal `sig` subsequently occurs, the signal is restored to its default behaviour and the `handler` is called. If the `handler` returns, execution resumes where the signal occurred. The initial state of the signals is implementation-defined.

When you install a signal handler within EiC you will most likely be overriding one of EiC's own internal signal handling routines:

```
EiC 1> #include <signal.h>
         (void)
EiC 2> raise(SIGFPE);

EiC maths  exception, file ::EiC::, line 2
EiC::Reset Local Stack Pointer
EiC: error clean up entry pt 0,1,2,3,4,
```

EiC assigns handlers for the following signals: `SIGBUS, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGUSR1`. It does this to keep the flow of an EiC interactive session going. That is, it prevents your code from causing EiC to abort in an undignified manner. While, in none-interactive mode it is no big deal if you override one of EiC's internal signal handlers, since you are saying that you will be handling that signal, but in an interactive session, things are different. You load translation units (§ 3.1, pg: 49), execute them, and various translation units may have no relationship to each other. Therefore, when you assign a new signal handler you should keep track of the initial one and reset it when appropriate:

```
EiC 1> #include <signal.h>
        (void)
EiC 2> void foo(int sig) { printf("my handle\n"); }
```

```
        (void)
EiC 3> void (*oldhandle)(int) = signal(SIGFPE,foo);
        (void)
EiC 4> raise(SIGFPE);
SIGFPE passed
        0
EiC 5> signal(SIGFPE,oldhandle); // reestablish old handle
        0x80babe8
EiC 6> raise(SIGFPE);

EiC maths  exception, file ::EiC::, line 6
EiC::Reset Local Stack Pointer
EiC: error clean up entry pt 0,1,2,3,4,
```

raise: Synopsis:

```
#include <signal.h>
int raise(int sig);
```

Send signal `sig` to the program. Non-zero returned if unsuccessful.
As an example program try `examples/sig1.c`

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void tick(int i) { printf("tick\n"); return;}
void tock(int i) { printf("tock\n"); return;}
int main()
{
    int i = 0, cnt = 1;
    while(1) {
        signal(SIGINT,tick); // note you must reestablish the handle
        sleep(1);
        raise(SIGINT);

        signal(SIGINT,tock);
        sleep(1);
        raise(SIGINT);
        if(i++==cnt)
            break;
    }
    signal(SIGINT,SIG_DFL);  // reset
    return 0;
```

```
}
```

Which should out put:

```
%> eic examples/sig1.c
tick
tock
tick
tock
```

4.1.9 stdarg.h

Defines macros that support functions with variable argument lists.

va_list: A type used to hold the information needed by the macros defined in `<stdarg.h>`.

va_start: `Synopsis`:

```
#include <stdarg.h>
void va_start(va_list ap, lastarg);
```

Initialisation macro to be called once, and before any unnamed argument is accessed. The argument `ap` must be declared as a local variable, and `lastarg` is the last named parameter in the controlling function's parameter list.

va_arg: `Synopsis`:

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Produce a value of the type (`type`) and corresponding to the next unnamed argument. It modifies the value of `ap`.

va_end: `Synopsis`:

```
#include <stdarg.h>
void va_end(va_list ap);
```

Must be called once, generally after all arguments have been processed, but definetly before function exit.

4.1.10 stddef.h

The header `<stddef.h>` declares the following macros and types:

ptrdiff_t: An implementation defined signed integral type, which represents the type of the result of subtracting two pointers.

size_t: An unsigned integral type, which is the return type from the operator `sizeof`.

NULL: Implementation defined null pointer constant.

offsetof: `offsetof(`*type*`,` *member-designator*`)`
Expands to type `size_t`, representing the offset in bytes of the structure member *member-designator* from the start of the structure *type*.

wchar_t: An integral type that can represent all values for any extended character in the set supported by `locales`.

4.1.11 stdio.h

`stdio.h` has the following types and macros defined:

FILE: Type which records information necessary to control a stream.

fpos_t: Variable used for specification of positions within an opened file.

size_t: See `<stddef.h>` § 4.1.10, pg: 117

stdin: Standard input stream. Automatically opened when a program begins execution.

stdout: Standard output stream. Automatically opened when a program begins execution.

stderr: Standard error stream. Automatically opened when a program begins execution.

FILENAME_MAX: Maximum permissible length of a file name

FOPEN_MAX: Maximum number of files which may be open simultaneously.

TMP_MAX: Maximum number of temporary files during program execution.

NULL: See `<stddef.h>` § 4.1.10, pg: 117

_IOFBF:

_IOLBF:

_IONBF: Macros used for the third argument to function `setvbuf`.

BUFSIZ: The default buffer size used by `setbuf`.

EOF: Macro used to indicate the end-of-file.

L_tmpnam: Macro that expands to an intergergral constant expression, which is the size for the array of characters allocated to the default name returned by function `tmpnam`.

SEEK_SET:

SEEK_CUR:

SEEK_END: Macros values used by `fseek` to locate current file seek position with respect to the beginning of the file, the current file position, or the end of the file respectively.

The following functions are defined in `stdio.h`

fopen: `Synopsis`

```
#include <stdio.h>
FILE* fopen(const char* filename, const char* mode);
```

Opens file `filename` and returns a pointer to an opened stream, or NULL on failure. The stream can be opened with `mode`:

```
"a"    Append. The file is created if it does not exist.
"w"    Write. If the file exists, it is deleted first.
"r"    Read. The file must already exist.
"r+"   Read and write. The file must already exist.
"w+"   Read and write. If the file exists, it is deleted
       first.
"a+"   Read and append. The file is created if it does  not
       exist.
"ab"   Append binary. The file is created if it does not
       exist.
"rb"   Open binary file for reading.
"wb"   Write binary file. If the files exist, it gets truncated
       to  zero first.
"ab+" or "a+b"  Append binary update.
"rb+" or "r+b"  Read binary update.
"wb+" or "w+b"  Write binary update.
```

freopen: Synopsis

```
#include <stdio.h>
FILE* freopen(const char* filename,
              const char* mode,
              FILE* stream);
```

Opens file `filename` with the specified `mode` and associates with it the specified `stream`. Returns stream or `NULL` on error. Usually used to change files associated with `stdin`, `stdout`, `stderr`.

fflush: Synopsis

```
#include <stdio.h>
int fflush(FILE* stream);
```

Flushes stream `stream`. Effect undefined for input `stream`. Returns `EOF` for write error, zero otherwise. `fflush(NULL)` flushes all output streams.

fclose: Synopsis

```
#include <stdio.h>
int fclose(FILE* stream);
```

Closes stream `stream` (after flushing, if output stream). Returns `EOF` on error, zero otherwise.

remove: Synopsis

```
#include <stdio.h>
int remove(const char* filename);
```

Removes file `filename`. Returns non-zero on failure.

rename: Synopsis

```
#include <stdio.h>
int rename(const char* oldname, const char* newname);
```

Changes name of file `oldname` to `newname`. Returns non-zero on failure.

tmpfile: Synopsis

```
#include <stdio.h>
FILE* tmpfile();
```

Creates temporary file (mode "wb+") which will be removed when closed or on normal program termination. Returns stream or NULL on failure.

tmpname: Synopsis

```
#include <stdio.h>
char* tmpname(char s[L_tmpnam]);
```

Assigns to `s` and returns unique name for temporary file.

setvbuf: Synopsis

```
#include <stdio.h>
int setvbuf(FILE* stream, char* buf, int mode, size_t size);
```

Controls buffering for stream `stream` and can only be used after the stream pointer by `stream` has been associated initially with an open file and before any read or write operations are performed. The argument `mode` determines how `stream` will be buffered such as `_IOLB`, `_IOFBF`, `_IONBF`. If `buf` is non-NULL then setvbuf will assign it as the buffer for `stream` otherwise `setvbuf` will allocate one and the value at `size` will determine the size of the buffer.

Returns zero on success or nonzero on error.

setbuf: Synopsis

```
#include <stdio.h>
void setbuf(FILE* stream, char* buf);
```

Controls buffering for stream `stream`. See also `setvbuf` 120.

fprintf: Synopsis

```
#include <stdio.h>
int fprintf(FILE* stream, const char* format, ...);
```

Converts (with format `format`) and writes output to stream `stream`. Number of characters written [negative on error] is returned. Between

```
Flags:
- ........ left adjust
+ ........ always sign
space .... outputs a space if the first character is not a
           sign.
0 ........ zero pad
# ........ Alternate form: for conversion character o,
           first digit will be zero, for [xX], prefix 0x
           or 0X to  non-zero, for [eEfgG], always decimal
           point, for [gG] trailing zeros not removed.
Width:
Period:
Precision:
  for conversion character s, maximum characters to be
  printed from the string, for [eEf], digits after decimal
  point, for [gG], significant digits, for an integer,
  minimum  number of digits to be printed.
Length modifier:
h ............. short or unsigned short
l ............. long or unsigned long
L ............. long double
Conversions:
d, i .......... int; signed decimal notation
o    .......... int; unsigned octal notation
x,X  .......... int; unsigned hexadecimal notation
u    .......... int; unsigned decimal notation
c    .......... int; single character
s    .......... char* ; outputs the character of a string
f    .......... double; [-]mmm.ddd
e,E  .......... double; [-]m.dddddde(+|-)xx
g,G  .......... double
p    .......... void*; print as pointer
n    .......... int*; number of chars written into arg
%    .......... print %
```

Example Uses of of the format string in `fprintf`:

```
%3d     print in a 3 digit field, right justified
%3.0f   print no decimal point and no fraction
%3.1f   print 1 digit after the decimal point
%.1f    print 1 digit after the decimal point, any width
```

Between the to specify left adjustment of the field, and two digit strings separated by a period. The first string specifies minimum field width, and the second string specifies the

maximum number of chars to be printed from the string.

```
:%10s:          :hello, world:
:%-10s:         :hello, world:
:%20s:          :        hello, world:
:%-20s:         :hello, world        :
:%20.10s:       :          hello, wor:
:%-20.10s:      :hello, wor          :
:%.10s:         :hello, wor:
```

printf: Synopsis

```
#include <stdio.h>
int printf(const char* format, ...);
```

`printf(f, ...)` is equivalent to `fprintf(stdout, f, ...)`

sprintf: Synopsis

```
#include <stdio.h>
int sprintf(char* s, const char* format, ...);
```

Like `fprintf`, but output written into string `s`, which must be large enough to hold the output, rather than to a stream. Output is null terminated; that is, the null character. Return length does not include the null terminating character.

vfprintf: Synopsis

```
#include <stdio.h>
int vfprintf(FILE* stream, const char* format, va_list arg);
```

Equivalent to `fprintf` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

vprintf: Synopsis

```
#include <stdio.h>
int vprintf(const char* format, va_list arg);
```

Equivalent to `printf` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See verb+¡stdarg.h¿+

vsprintf: Synopsis

```
#include <stdio.h>
int vsprintf(char* s, const char* format, va_list arg);
```

Equivalent to `sprintf` except that the variable argument list is replaced by `arg`, which must have been initialised by the `va_start` macro and may have been used in calls to `va_arg`. See `<stdarg.h>`

fscanf: Synopsis

```
#include <stdio.h>
int fscanf(FILE* stream, const char* format, ...);
```

Performs formatted input conversion, reading from stream `stream` according to format `format`. The function returns when `format` is fully processed. Returns `EOF` if end-of-file or error occurs before any conversion; otherwise, the number of items converted and assigned. Each of the arguments following format must be a pointer. Format string may contain:

- Blanks, Tabs : ignored
- ordinary characters : expected to match next non-white-space
- % : Conversion specification, consisting of %, optional assignment suppression character *, optional number indicating maximum field width, optional [hlL] indicating width of target, conversion character.

Conversion characters:

d
: decimal integer; int* parameter required

i
: integer; int* parameter required; decimal, octal or hex

o
: octal integer; int* parameter required

u
: unsigned decimal integer; unsigned int* parameter required

x
: hexadecimal integer; int* parameter required

c
: characters; char* parameter required; up to width; no '\0' added; no skip

s
: string of non-white-space; char* parameter required; '\0' added

e,f,g
: floating-point number; float* parameter required

p
: pointer value; void* parameter required

n
: chars read so far; int* parameter required

[...]
: longest non-empty string from set; char* parameter required; '\0'

[^...]
: longest non-empty string not from set; char* parameter required; '\0'

%
: literal %; no assignment

scanf: Synopsis

```
#include <stdio.h>
int scanf(const char* format, ...);
```

`scanf(f, ...)` is equivalent to `fscanf(stdin, f, ...)`

sscanf: Synopsis

```
#include <stdio.h>
int sscanf(char* s, const char* format, ...);
```

Like `fscanf`, but input read from string `s`.

fgetc: Synopsis

```
#include <stdio.h>
int fgetc(FILE* stream);
```

Returns next character from stream `stream` as an `unsigned char`, or `EOF` on end-of-file or error.

fgets: Synopsis

```
#include <stdio.h>
char* fgets(char* s, int n, FILE* stream);
```

Reads at most the next `n`-1 characters from stream stream into `s`, stopping if a newline is encountered (after copying the newline to s). s is null terminated. Returns `s`, or `NULL` on end-of-file or error.

fputc: Synopsis

```
#include <stdio.h>
int fputc(int c, FILE* stream);
```

Writes `c`, converted to `unsigned char`, to stream `stream`. Returns the character written, or `EOF` on error.

fputs: Synopsis

```
#include <stdio.h>
char* fputs(const char* s, FILE* stream);
```

Writes `s`, which need not contain `'\n'` on stream stream. Returns non-negative on success, `EOF` on error.

getc: Synopsis

```
#include <stdio.h>
int getc(FILE* stream);
```

Equivalent to `fgetc` except that it may be a macro.

getchar: Synopsis

```
#include <stdio.h>
int getchar();
```

Equivalent to `getc(stdin)`.

gets: Synopsis

```
#include <stdio.h>
char* gets(char* s);
```

Reads next line from `stdin` into `s`. Replaces terminating newline with `'\0'`. Returns `s`, or `NULL` on end-of-file or error.

putc: Synopsis

```
#include <stdio.h>
int putc(int c, FILE* stream);
```

Equivalent to `fputc` except that it may be a macro.

putchar: Synopsis

```
#include <stdio.h>
int putchar(int c);
```

`putchar(c)` is equivalent to `putc(c, stdout)`.

puts: Synopsis

```
#include <stdio.h>
int puts(const char* s);
```

Writes `s` and a newline to `stdout`. Returns non-negative on success, `EOF` on error.

unget: Synopsis

```
#include <stdio.h>
int unget(int c, FILE* stream);
```

Pushes `c` (which must not be EOF), converted to `unsigned char`, onto stream `stream` such that it will be returned by the next read. Only one character of pushback is guaranteed for a stream. Returns `c`, or `EOF` on error.

fread: Synopsis

```
#include <stdio.h>
size_t fread(void* ptr,
             size_t size,
             size_t nobj,
             FILE* stream);
```

Reads at most `nobj` objects of size `size` from stream `stream` into `ptr`. Returns the number of objects read. `feof` and `ferror` must be used to determine status.

fwrite: Synopsis

```
#include <stdio.h>
size_t fwrite(const void* ptr,
              size_t size,
              size_t nobj,
              FILE* stream);
```

Writes to stream `stream`, `nobj` objects of size `size` from array `ptr`. Returns the number of objects written (which will be less than `nobj` on error).

fseek: Synopsis

```
#include <stdio.h>
int fseek(FILE* stream, long offset, int origin);
```

Sets file position for stream `stream`. For a binary file, position is set to `offset` characters from `origin`, which may be `SEEK_SET` (beginning), `SEEK_CUR` (current position) or `SEEK_END` (end-of-file); for a text stream, `offset` must be zero or a value returned by `ftell` (in which case `origin` must be `SEEK_SET`). Returns non-zero on error.

ftell: Synopsis

```
#include <stdio.h>
long ftell(FILE* stream);
```

Returns current file position for stream `stream`, or -1L on error.

rewind: Synopsis

```
#include <stdio.h>
void rewind(FILE* stream);
```

`rewind(stream)` is equivalent to `fseek(stream, 0L, SEEK_SET)`;

fgetpos: Synopsis

```
#include <stdio.h>
int fgetpos(FILE* stream, fpos_t* ptr);
```

Assigns current position in stream stream to `*ptr`. Type `fpos_t` is suitable for recording such values. Returns non-zero on error.

fsetpos: Synopsis

```
#include <stdio.h>
int fsetpos(FILE* stream, const fpos_t* ptr);
```

Sets current position of stream `stream` to `*ptr`. Returns non-zero on `error`.

clearerr: Synopsis

```
#include <stdio.h>
void clearerr(FILE* stream);
```

Clears the end-of-file and error indicators for stream `stream`.

feof: Synopsis

```
#include <stdio.h>
int feof(FILE* stream);
```

Returns non-zero if end-of-file indicator for stream `stream` is set.

ferror: Synopsis

```
#include <stdio.h>
int ferror(FILE* stream);
```

Returns non-zero if error indicator for stream `stream` is set.

perror: Synopsis

```
#include <stdio.h>
void perror(const char* s);
```

Prints `s` and implementation-defined error message corresponding to `errno`: `fprintf(stderr, "%s: %s\n", s, "error message")`
See `strerror`.

4.1.12 stdlib.h

The header file `<stdlib.h>` contains the following types and macros:

RAND_MAX: Integral constant, which is the maximum value returned from `rand`.

EXIT_FAILURE:

EXIT_SUCCESS: Macros defined for successful or unsuccessful program termination.

size_t: See `<stddef.h>` § 4.1.10, pg: 117

NULL: See `<stddef.h>` § 4.1.10, pg: 117

div_t: A structure type as returned by the `div` function.

ldiv_t: A structure type as returned by the `ldiv` function.

wchar_t:

The following functions are defined in the header `<stdlib.h>`:

atof: Synopsis:

```
#include <stdlib.h>
float atof(const char *s);
```

Converts the string of ASCII characters, which represent a decimal number to a float. The string consists of optional leading spaces or tabs, an optional plus or minus sign `(+ or -)` followed by one or more decimal digits. Returns the value of the ASCII number string. The string passed to atof can contain a decimal point with digits to the right of the decimal point. It can also take the form of a floating point constant.

atoi: Synopsis:

```
#include <stdlib.h>
int atoi(const char* s);
```

Returns numerical value of `s`. Equivalent to `(int)strtol(s,NULL,10)`.

atol: Synopsis:

```
#include <stdlib.h>
long atol(const char* s);
```

Returns numerical value of s. Equivalent to `strtol(s, NULL, 10)`.

strtod: Synopsis:

```
#include <stdlib.h>
double strtod(const char* s, char** endp);
```

Converts prefix of `s` to double, ignoring leading white spaces. Stores a pointer to any unconverted suffix in *endp if endp is non-NULL. In the case of overflow, `HUGE_VAL` is returned with the appropriate sign; for the case of underflow, zero is returned. In either case, errno is set to `ERANGE`.

strtol: Synopsis:

```
#include <stdlib.h>
long strtol(const char* s, char** endp, int base);
```

Converts prefix of `s` to long, ignoring leading white spaces. Stores a pointer to any unconverted suffix in `*endp` if `endp` is non-NULL. If base between 2 and 36, that base used; if zero, leading 0X or 0x implies hexadecimal, a leading 0 implies octal, otherwise a decimal conversion is used. Leading 0X or 0x permitted for base 16. In the case of overflow, `LONG_MAX` or for the case of underflow `LONG_MIN` is returned and `errno` is set to ERANGE.

strtoul: Synopsis:

```
#include <stdlib.h>
unsigned long strtoul(const char* s, char** endp, int base);
```

As for `strtol` except result is unsigned long and in the case of overflow `ULONG_MAX` is returned.

rand: Synopsis:

```
#include <stdlib.h>
int rand();
```

Returns pseudo-random number in range 0 to `RAND_MAX`.

srand: Synopsis:

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Uses `seed` as `seed` for new sequence of pseudo-random numbers. The defautl initial value for `seed` is 1.

calloc: Synopsis:

```
#include <stdlib.h>
void* calloc(size_t nobj, size_t size);
```

Returns pointer to zero-initialised newly-allocated space for an array of `nobj` objects each of size `size`, or `NULL` if request cannot be satisfied.

malloc: Synopsis:

```
#include <stdlib.h>
void* malloc(size_t size);
```

Returns pointer to uninitialised newly-allocated space for an object of size `size`, or `NULL` if request cannot be satisfied.

realloc: Synopsis:

```
#include <stdlib.h>
void* realloc(void* p, size_t size);
```

Changes the size of the object to which `p` points to `size`. Contents unchanged to minimum of old and new sizes. If new size larger, new space is uninitialised. Returns pointer to the new space or, if request cannot be satisfied NULL leaving p unchanged.

free: Synopsis:

```
#include <stdlib.h>
void free(void* p);
```

Deallocates space to which `p` points. If `p` is NULL there is no effect; otherwise it must be a pointer returned by `calloc`, `malloc` or `realloc`.

abort: `Synopsis`:

```
#include <stdlib.h>
void abort();
```

Causes program to terminate abnormally, as if by `raise(SIGABRT)`.

`item`[exit] `Synopsis`:

```
#include <stdlib.h>
void exit(int status);
```

Causes normal program termination. Functions installed using `atexit` are called in reverse order of registration. Open files are flushed and open streams are closed and control is returned to environment. The value of `status` is returned to environment in an implementation-dependent manner. Zero indicates successful termination and the values `EXIT_SUCCESS` and `EXIT_FAILURE` may also be used.

atexit: `Synopsis`:

```
#include <stdlib.h>
int atexit(void (*fcm)(void));
```

Registers `fcm` to be called, in reverse order, when the program terminates or via a call to Texit. Returns zero on success else a non-zero value is returned.

system: `Synopsis`:

```
#include <stdlib.h>
int system(const char* s);
```

Passes `s` to environment for execution. If `s` is `NULL`, non-zero returned if command processor exists; return value is implementation-dependent if `s` is non-NULL.

getenv: `Synopsis`:

```
#include <stdlib.h>
char* getenv(const char* name);
```

Returns (implementation-dependent) environment string associated with `name`, or `NULL` if no such string exists.

puttenv: `Synopsis`:

```
#include <stdlib.h>
int putenv(const char* name);
```

Accepts a string in the form *name=value* and inserts it into the system environment list, and if needed replacing any previous definition.

Returns 0 on success or -1 on error. Errors: `ENOMEM` insufficient space to allocate new environment.

besearch: Synopsis:

```
#include <stdlib.h>
void* bsearch(const void* key,
              const void* base,
              size_t n,
              size_t size,
              int (*cmp)(const void* keyval,
                         const void* datum));
```

Searches `base[0]...base[n-1]` for item matching *`key`. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater. The n items of base must be in ascending order. Returns a pointer to the matching entry or `NULL` if not found.

qsort: Synopsis:

```
#include <stdlib.h>
void qsort(void* base,
           size_t n,
           size_t size,
           int (*cmp)(const void *a1, const void * a2));
```

Arranges into ascending order the array `base[0]...base[n-1]` of objects of size size. Comparison function `cmp` must return negative if first argument is less than second, zero if equal and positive if greater.

abs: Synopsis:

```
#include <stdlib.h>
int abs(int n);
```

Returns absolute value of `n`.

labs: Synopsis:

```
#include <stdlib.h>
long labs(long n);
```

Returns absolute value of `n`.

div: Synopsis:

```
#include <stdlib.h>
div_t div(int num, int denom);
```

Returns in fields `quot` and `rem` of structure of type `div_t` the quotient and remainder of num/denom respectively.

ldiv: Synopsis:

```
#include <stdlib.h>
ldiv_t ldiv(long num, long denom);
```

Returns in fields `quot` and `rem` of structure of type `ldiv_t` the quotient and remainder of num/denom respectively.

4.1.13 string.h

The header file `<string.h>` defines the following types and macros:

size_t: See `<stddef.h>` § 4.1.10, pg: 117

NULL: See `<stddef.h>` § 4.1.10, pg: 117

The following functions are defined in `<string.h>`:

strcpy: Synopsis:

```
#include <string.h>
char* strcpy(char* s, const char* ct);
```

Copy `ct` to `s` including terminating null character. Returns a pointer to `s`.

strncpy: Synopsis:

```
#include <string.h>
char* strncpy(char* s, const char* ct, int n);
```

Copy at most `n` characters of `ct` to `s`. Pad with zeros if `ct` is of length less than `n`. Returns a pointer to `s`.

strcat: Synopsis:

```
#include <string.h>
char *strcat(char *s2, const char *s1);
```

Concatenates the string pointed to by `s2` to the string pointed to by `s1`. The calling program must assure that `s1` has enough space for the concatenation.

strncat: Synopsis:

```
#include <string.h>
char* strncat(char* s, const char* ct, int n);
```

Concatenate at most `n` characters of `ct` to `s`. Terminate `s` with the null character and returns a pointer to it.

strcmp: Synopsis:

```
#include <string.h>
int strcmp(const char* s1, const char* s2);
```

Compares two strings. The comparison stops when a `null` terminator is encountered in either of the two strings. Returns a 0 if the two strings are identical, less than zero if `s2` is greater than `s1`, and greater than zero if `s1` is greater than `s2`.

strdup: Synopsis:

```
#include <string.h>
char * strdup(const char* s);
```

Returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc, and can be freed with free.

strncmp: Synopsis:

```
#include <string.h>
int strncmp(const char* s1, const char* s2, int n);
```

Compares two strings. The comparison stops when a `null` terminator is encountered in either of the two strings or when n number of bytes are compared. Returns a 0 if the two strings are identical, less than zero if `s2` is greater than `s1`, and greater than zero if `s1` is greater than `s2`.

strchr: Synopsis:

```
#include <string.h>
char* strchr(const char* s1, int c);
```

Return pointer to first occurrence of `c` in `s1`, or `NULL` if not found.

strrchr: Synopsis:

```
#include <string.h>
char* strrchr(const char* s1, int c);
```

Return pointer to last occurrence of `c` in `s1`, or `NULL` if not found.

strspn: Synopsis:

```
#include <string.h>
size_t strspn(const char* s1, const char* s2);
```

Return length of prefix of `s1` consisting entirely of characters in `s2`.

strcspn: Synopsis:

```
#include <string.h>
size_t strcspn(const char* s1, const char* s2);
```

Return length of prefix of `s1` consisting entirely of characters not in `s2`.

strpbrk: Synopsis:

```
#include <string.h>
char* strpbrk(const char* s1, const char* s2);
```

Return pointer to first occurrence within `s1` of any character of `s2`, or `NULL` if not found.

strstr: Synopsis:

```
#include <string.h>
char* strstr(const char* s1, const char* s2);
```

Return pointer to first occurrence of `s2` in `s1`, or `NULL` if not found.

strlen: Synopsis:

```
#include <string.h>
size_t strlen(const char* s1);
```

Return length of `s1`.

strerror: Synopsis:

```
#include <string.h>
char* strerror(int n);
```

Return pointer to implementation-defined string corresponding with error `n`.

strtok: Synopsis:

```
#include <string.h>
char* strtok(char* s, const char* ct);
```

A sequence of calls to `strtok` returns tokens from `s` delimted by a character in `ct`. A non-NULL `s` indicates the first call in a sequence. Also,`ct` may differ on each call. Returns `NULL` when no such token found.

memcpy: Synopsis:

```
#include <string.h>
void* memcpy(void* dest, const void* src, int n);
```

Copy `n` characters from `src` to `dest`. Return `dest`. Does not work correctly if objects overlap.

memmove: Synopsis:

```
#include <string.h>
void* memmove(void* dest, const void* src, int n);
```

Copy `n` characters from `src` to `dest`. Return `dest`. Works correctly even if objects overlap.

memcmp: Synopsis:

```
#include <string.h>
int memcmp(const void* s1, const void* s2, int n);
```

Compare first `n` characters of `s1` with `s2`. Return negative if `s1` ¡ `s2`, zero if `s1` == `s2`, positive if `s1` ¿ `s2`.

memchr: Synopsis:

```
#include <string.h>
void* memchr(const char* s1, int c, int n);
```

Return pointer to first occurrence of `c` in the first `n` characters of `s1`, or `NULL` if not found.

memset: Synopsis:

```
#include <string.h>
void* memset(char* s, int c, int n);
```

Replace each of the first `n` characters of `s` by `c`. Return `s`.

4.1.14 time.h

The header `<time.h>` declares the following macros and types:

clock_t: An arithmetic type representing time.

time_t: An arithmetic type representing time.

CLOCKS_PER_SEC: The number of `clock_t` units per second.

struct tm: Represents the components of calendar time:

```
int tm_sec; /* seconds after the minute */
int tm_min; /* minutes after the hour */
int tm_hour; /* hours since midnight */
int tm_mday; /* day of the month */
int tm_mon; /* months since January */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday */
int tm_yday; /* days since January 1 */
int tm_isdst; /* Daylight Saving Time flag */
```

The value of `tm_isdst` is positive if Daylight saving time is in effect, zero if not in effect, negative if information unavailable.

The following functions are defined in `<time.h>`:

clock: Synopsis:

```
#include <time.h>
clock_t clock(void);
```

Returns processor time used by program or -1 if not available.

time: Synopsis:

```
#include <time.h>
time_t time(time_t* tp);
```

Returns current calendar time or -1 if not available. If `tp` is non-NULL, return value is also assigned to *`tp`.

difftime: Synopsis:

```
#include <time.h>
double difftime(time_t time2, time_t time1);
```

Returns the difference is seconds between `time2` and `time1`.

mktime: Synopsis:

```
#include <time.h>
time_t mktime(struct tm* tp);
```

Returns the local time corresponding to *`tp`, or -1 if it cannot be represented.

asctime: Synopsis:

```
#include <time.h>
char* asctime(const struct tm* tp);
```

Returns the given time as a string of the form: `Sun Jan 3 14:14:13 1988\n\0`

ctime: Synopsis:

```
#include <time.h>
char* ctime(const time_t* tp);
```

Converts the given calendar time, `tp`, to a local time and returns the equivalent string. Equivalent to: asctime(localtime(tp))

gmtime: Synopsis:

```
#include <time.h>
struct tm* gmtime(const time_t* tp);
```

Returns the given calendar time converted into Coordinated Universal Time, or `NULL` if not available.

localtime: Synopsis:

```
#include <time.h>
struct tm* localtime(const time_t* tp);
```

Returns calendar time *`tp` converted into local time.

strftime: Synopsis:

```
#include <time.h>
size_t strftime(char* s,
                size_t smax,
                const char* fmt,
                const struct tm* tp);
```

Formats *`tp` into s according to `fmt`.

Notes: Local time may differ from calendar time, for example because of time zone.

4.2 POSIX.1 library support

Here is EiC's current implementation of the POSIX.1 library. It is by no means complete, but rather a strict subset. For those interested, the POSIX.1 environment is more formally presented by (Zlotnick, 1991) or (Stevens, 1992).

The test macro `_POSIX_SOURCE` is used and as documented in the IEEE POSIX.1 standard, where the programmer is required to define the `_POSIX_SOURCE` feature test macro to obtain the `POSIX.1` namespace and `POSIX.1` functionality.

This macro can be defined, at compile time `(-D_POSIX_SOURCE)` or by using `#define` directives in the source files before any `#include` directives:

```
#define  _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
```

It is only needed for those header shared between `POSIX.1` and `ISO-C`, when the `POSIX.1` features are to be made visible (header files that are underlined are currently not supported):

ISO-C	POSIX.1	IS0-C	POSIX.1
assert.h		stdio.h	stdio.h
ctype.hr		stdlib.h	
	dirent.h	string.h	
errno.h	errno.h		sys/stat.h
	fcntl.h		sys/times.h
float.h			sys/utsname.h
	grp.h		sys/wait.h
iso646.h			termios.h
limits.h	limits.h	time.h	time.h
locale.h			unistd.h
math.h			utime.h
setjmp.h	setjmp.h	wchar.h	
signal.h	signal.h	wctype.h	
stdarg.h			
stddef.h			

4.2.1 dirent.h

The EiC header file `<dirent.h>` contains objects, types and functions for reading and opening directories. To make or remove a directory see section § 4.2.8, pg: 145. Also,

while anyone with the appropriate access permissions may read a directory, only the kernel can write to a directory. Then `<dirent.h>` defines the following type and structure:

DIR: A directory stream is represented by the type `DIR`, which is similar to the `<stdio.h>` type `FILE` (page 118).

struct dirent: While the the dirent structure is implementation dependent, it will contain at least:

```
ino_t d_ino;  /* inode number of entry */
char d_name[NAME_SIZE + 1]; /* name (null-terminated) */
```

Note the size of `d_name` is also implementation dependent. The `dirent struct` specifies the structure type that is used to hold information about individual directory entries, such as files etc.

The following functions are defined in `<dirent.h>`:

closedir: `Synopsis`:

```
#include <sys/types.h>
#include <dirent.h>
int closedir(DIR *dirp);
```

Closes the directory stream associated with `dirp`. Returns 0 on success, or -1 on error and sets `errno` to `EBADF`.

opendir: `Synopsis`:

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *dirname);
```

Opens the directory stream associated with the directory `dirname`, and returns a pointer to the opened stream. The directory will be opened such that, the stream pointer is positioned at the first entry in the directory. Returns a pointer to the directory on success, else `NULL` on error and will set `errno` to one of: `EACESS`, `EMFILE`, `ENOENT`, `ENFILE`, `ENOMEM` or `ENOTDIR`.

readdir: `Synopsis`:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

Reads the next dirent structure from the stream `dirp`. Returns a pointer to the associated `struct dirent`, else `NULL` if the end-of-file mark has been reached or on an error, and in which case it sets `errno` to `EBADF`.

rewinddir: `Synopsis`:

```
#include <sys/types.h>
#include <dirent.h>
void rewinddir(DIR *dirp);
```

Rewinds or resets the directory stream `dirp` back to the beginning. No error codes used.

4.2.2 errno.h

The header file `<errno.h>` has already been discussed with respect to the ISO C library specifications (see § 4.1.3, pg: 109). Here its POSIX.1 additions are report.

The EiC header file `<errno.h>` defines the following extra macros:

E2BIG:	Argument list too long.
EACCES:	Permission denied.
EAGAIN:	Device or resource unavailable; try again later.
EBADF:	Bad file descriptor.
EBUSY:	Device or resource busy.
ECHILD:	No child processes.
EDEADLK:	Resource deadlock would result.
EEXIST:	File exists.
EFAULT:	Bad address.
EFBIG:	File too large.
EINTR:	Function interrupted system call.
EINVAL:	Invalid argument.
EIO:	I/O error.
EISDIR:	Is a directory.
EMFILE:	Too many open files.
EMLINK:	Too many links.
ENAMETOOLONG:	File name too long.
ENFILE:	File table overflow because of too many open files.
ENODEV:	No such device.
ENOENT:	No such file or directory.
ENOEXEC:	Executable format error, because file is not executable.
ENOLCK:	No record locks available.
ENOMEM:	Out of memory.
ENOSPC:	No space left on device.
ENOSYS:	Function not implemented or supported
ENOTDIR:	Not a directory.
ENOTEMPTY:	Directory not empty.
ENOTTY:	Not a typewriter or inappropriate I/O control operation.

ENXIO: No such device or address.
EPERM: Operation not permitte
EPIPE: Broken pipe.
EROFS: Read-only file system.
ESPIPE: Illegal seek operation.
ESRCH: No such process.
EXDEV: Cross-device link; invalid link.

4.2.3 fcntl.h

The EiC header file `<fcntl.h>` defines the following macros:

O_APPEND: If on, set offset to end-of-file before each write.
O_CREAT: If file does not exist, the the file is created and with file attribute according to the value of *mode*. If files does exist, then this flag has no effect.
O_EXCL: Fail if file exists and if `O_CREAT` is also specified. Otherwise, create the file.
O_NOCTTY: Not used with regular files.
O_NONBLOCK: Not used with regular files.
O_RDONLY: Open for read only.
O_RDWR: Open for read and write.
O_TRUNC: If the file exists, its length will be truncated to zero.
O_WRONLY: Open for write only.
O_NDELAY: For compatibility with System V.3.
O_BINARY: Open file in binary mode. Added for DOS compatibility.
O_TEXT: Open file in text mode. Added for DOS compatibility.

The following functions are defined in `<fcntl.h>`:

creat: Synopsis:

```
#include <sys/types.h>
#include <sys/stats.h>
#include <fcntl.h>
int creat(const char *path,  mode_t mode);
```

Creates a new file or rewrites an existing one for writing, as specified by `path`. Its access is specified by `mode`, which maybe one or a bitwise combination of: `S_IS[UG]ID`, `S_ISVTX`, `S_I[RWX](GRP|USR|OTH)`. It returns a nonnegative file descriptor if successful, else it returns -1 and sets `errno` to one of: `EACCES`, `EEXIST`, `EINTR`, `EISDIR`, `EMFILE`, `ENAMETOOLONG`, `ENFILE`, `ENOENT`, `ENOSPS`, `ENOTODIR`, or `EROFS`.

open: Synopsis:

```
#include <sys/types.h>
#include <sys/stats.h>
#include <fcntl.h>
int open(const char *path, int access, ... /* mode_t mode */);
```

Create or open the file specified by `path`, with access defined by `access`, which maybe one or a bitwise combination of: `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_NONBLOCK`, `O_NOCTTY`, `O_RDONLY`, `O_RDWR` or `O_WRONLY`. The extra argument `mode` is used when creating a file with the access flag `O_CREAT` specified. The `mode` maybe one or a bitwise combination of `S_IS[UG]ID`, `S_ISVTX`, `S_I[RWX](GRP|USR|OTH)`. `Open` returns a nonnegative file descriptor if successful, else it returns -1 and sets `errno` to one of: `EACCES`, `EEXIST`, `EINTR`, `EISDIR`, `EMFILE`, `ENAMETOOLONG`, `ENFILE`, `ENOENT`, `ENOSPS`, `ENOTODIR`, TENXIO or `EROFS`.

fcntl: Synopsis:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int filedes, int cmd, ..., /* optional int arg */);
```

The properties of a file can be changed via the `fcntl` function. It is used for the following five purposes:

1. Makes arg be a copy of fd, closing fd first if necessary, when `cmd` = `F_DUPFD`.
2. Read or set file discriptor flags, `cmd` = `F_GETFD` or `F_SETFD`.
3. Read or set a file's status flags , `cmd` = `F_GETFL` or `F_SETFL`.
4. Read or set the process ID (or process group) of the owner of a socket, `cmd` = `F_GETOWN` or `F_SETOWN`.
5. Read or set record locks, `cmd` = `F_GETL`, `F_SETLK` or `SETLKW`.

All commands return -1 on error otherwise the return value depends on the input command: `F_DUPFD`, a new descriptor; `F_GETFD`, the value of the flag `F_GETFL` value of flags and `F_GETOWN` a positive or negative process `ID`.

On error, sets `errno` to one of: `EACCESS`, `EAGAIN`, `EDEADLK`.

4.2.4 limits.h

The header `<limits.h>` declares the following POSIX.I macros:

ARG_MAX: maximum length of argument to the 'exec' function

CHILD_MAX: maximum number of simultaneous processes per real user ID at any one time

LINK_MAX: number of links a file may have

MAX_CANON: size of the canonical input queue

MAX_INPUT: size of the type-ahead buffer

NAME_MAX: maximum number of bytes in a file name, not including null termination.

NGROUPS_MAX: maximum number of supplementary group IDs that one process can have.
OPEN_MAX: number of files that a single process can have open simultaneously
PIPE_BUF: maximum number of bytes written atomically to a pipe
SSIZE_MAX: largest value for object of type ssize_t
TZNAME_MAX: maximum bumber of bytes for the a time zone name.
_POSIX_ARG_MAX: Maximum length of arguments to 'execve', including environment.
_POSIX_CHILD_MAX: Maximum simultaneous processes per real user ID.
_POSIX_LINK_MAX: Maximum link count of a file.
_POSIX_MAX_CANON: Number of bytes in a terminal canonical input queue.
_POSIX_MAX_INPUT: Number of bytes for which space will be available in a terminal input queue.
_POSIX_NAME_MAX: Number of bytes in a filename.
_POSIX_NGROUPS_MAX: Number of simultaneous supplementary group IDs per process.
_POSIX_OPEN_MAX: Number of files one process can have open at once.
_POSIX_PATH_MAX: Number of bytes in a pathname.
_POSIX_PIPE_BUF: Number of bytes than can be written atomically to a pipe.
_POSIX_SSIZE_MAX: Largest value of a 'ssize_t'.
_POSIX_STREAM_MAX: Number of streams a process can have open at once.
_POSIX_TZNAME_MAX: Maximum length of a timezone name (element of 'tzname').
_POSIX_QLIMIT: Maximum number of connections that can be queued on a socket.
_POSIX_HIWAT: Maximum number of bytes that can be buffered on a socket for send or receive.
_POSIX_UIO_MAXIOV: Maximum number of elements in an 'iovec' array.
_POSIX_TTY_NAME_MAX: Maximum number of characters in a tty name.
_POSIX_LOGIN_NAME_MAX: Maximum length of login name

4.2.5 signal.h

The header `<signal.h>` declares the following POSIX.I macros:

SIGALRM: Alarm clock.
SIGCHLD: Child process terminated or stopped.
SIGHUP: Hangup.
SIGKILL: Kill (cannot be caught or ignored).
SIGPIPE: Write on a pipe with no one to read it.
SIGQUIT: Terminal quit signal.
SIGSTOP: Stop executing (cannot be caught or ignored).
SIGTSTP: Terminal stop signal.
SIGTTIN: Background process attempting read.
SIGTTOU: Background process attempting write.
SIGUSR1: User-defined signal 1.
SIGUSR2: User-defined signal 2.

4.2.6 sys/stat.h

The EiC header file `<sys/stats.h>` defines symbolic constants that are used when specifying the `mode_t` access of files. It defines the following macros and one structure specifier:

S_IRGRP: Group read permission.

S_IROTH: Other read permission.

S_IRUSR: Owner read permission. This is identical to the `S_IREAD` used by DOS.

S_IRWXG: Group read, write and execute permission.
`S_IRWXG = S_IRGRP | S_IWGRP | SI_XGRP`

S_IRWXO: Other read, write and execute permission.
`S_IRWXO = S_IROTH | S_IWOTH | S_IXOTH`

S_IRWXU: Owner read, write and execute permission.
`S_IRWXU = S_IRUSR | S_IWUSR | S_IXUSR`

S_ISBLK: Is block special file.

S_ISCHR: Is character special file.

S_ISDIR: Is directory.

S_ISFIFO: Is pipe or FIFO.

S_ISGID: set group id on execution

S_ISREG: set user id on execution

S_IWGRP: Group write permission.

S_IWOTH: Other write permission.

S_IWUSR: Owner write permission. This is identical to `S_IWRITE` used by DOS.

S_IXGRP: Group execute permission.

S_IXOTH: Other execute permission.

S_IXUSR: Owner execute permission.

struct stat: A file status attributes are easily collected into a structure as specified by `struct stat`, which has at least the following members:

```
mode_t  st_mode;    /* File mode */
ino_t   st_ino;     /* File serial number */
dev_t   st_dev;     /* File system device number */
nlink_t st_nlink;   /* Number of links */
uid_t   st_uid;     /* User ID of the file's owner */
gid_t   st_gid;     /* Group ID of the file's group */
off_t   st_size;    /* File size in bytes */
time_t  st_atime;   /* Time of last access */
time_t  st_mtime;   /* Time of last data modification */
time_t  st_ctime;   /* Time of last file status change */

/* ... */           /* other possible members */
```

The following functions are defined in `<sys/stat.h>`:

chmod: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char * path, mode_t mode);
```

Change access permission of the file specified by `path` to `mode`, which maybe one or a bitwise combination of `S_IS[UG]ID`, `S_ISVTX`, `S_I[RWX](GRP|USR|OTH)`. Returns zero on success or -1 on error and sets `errno` to one of: `EACCESS`, `ENAMETOOLONG`, `ENOTDIR`, `ENOENT`, `EPERM` OR `EROFS`.

fstat: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
int fstat(int filedes, struct stat *buf)
```

Gets the open file or directory information associate with `filedes`. It stores the information in the `stat` structure, pointed to by `buf`. It returns zero on success, else -1 on error and sets `errno` to `EBADF`.

mkdir: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *path, mode_t mode)
```

Creates a directory from the given path with access specified by `mode`, which maybe one or a bitwise combination of `S_IS[UG]ID`, `S_ISVTX`, `S_I[RWX](GRP|USR|OTH)`. Returns 0 on success, else -1 on error and then sets `errno` to one of: `EACCESS`, `ENAMETOOLONG`, `ENOENT` or `ENOTDIR`.

mkfifo: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo(const char * path, mode_t mode)
```

Creates a FIFO name pipe with access specified by `mode`, which maybe one or a bitwise combination of `S_IS[UG]ID`, `S_ISVTX`, `S_I[RWX](GRP|USR|OTH)`. Returns 0 on success, else -1 on error and sets `errno` to one of: `EACCESS`, `EEXIST`, `ENAMETOOLONG`, `ENOENT`, `ENOSPC`, `ENOTDIR` or `EROFS`.

stat: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *file_name, struct stat *buf)
```

The same as `fstat` above, but is applied to a file name rather than an already opened file. Returns 0 on success, else -1 on error and sets `errno` to one of: `EACCESS`, `ENAMETOOLONG`, `ENOENT` or `ENOTDIR`.

umask: Synopsis:

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask)
```

Sets the file mode creation mask for the calling process to `cmask`. Returns the previous creation mask. No error codes used.

4.2.7 sys/types.h

The EiC header file `<sys/types.h>` defines the following object types.

dev_t:	Device number (major and minor).
gid_t:	Group ID.
ino_t:	Numeric i-node value.
mode_t:	File type and creation mode.
nlink_t:	Number of links associated with a directory.
off_t:	For recording file offset position and file sizes.
pid_t:	Process ID number and process group ID¿
size_t:	See section § 4.1.10, pg: 117
ssize_t:	POSIX byte count.
uid_t:	User IDs.

4.2.8 unistd.h

The EiC header file `<unistd.h>` defines the following macros:

F_OK:	Does file exist.
W_OK:	Writable by caller.
R_OK:	Readable by caller.
X_OK:	Executable by caller.
STDIN_FILENO:	Standard input.
STDOUT_FILENO:	Standard output.
STDERR_FILENO:	Standard error output.

The following functions are presently implemented from `<unistd.h>` by EiC:

access: Synopsis

```
#include <unistd.h>
int access(const char * path, int mode);
```

Checks the file pointed to by `path` for accessibily according to `mode`, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. This allows a setuid process to verify that the user running it would have had permission to access this file. The `mode` can be `F_OK`, `R_OK`, `W_OK`, or `X_OK`. Returns zero on success or -1 on failure and sets `errno` to one of: EACCES, ENAMETOOLONG, ENOENT, or EROFS.

alarm: Synopsis

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

`alarm` arranges to have a `SIGALRM` delivered to the process for the signal `SIGALRM` after the specified number of seconds have elapsed. If `seconds` is zero then no new alarm is schededuled. In all cases the any previous `alarm`s are cancelled and `return`s the number of seconds remaining until any scheduled alarm was due to be delivered. `Return`s zero if there is no previously scheduled alarm.

chdir: Synopsis

```
#include <unistd.h>
int chdir(const char * pathname);
```

`chdir` changes the current working directory to the one specified in `pathname`. Returns on success, 0 and on error returns -1 and it will set `errno` to one of: `EFAULT` `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, `EACCES`, `ELOOP`, `EIO`, `EBADF` or `EACCES`

close: Synopsis

```
#include <unistd.h>
int close(int handle);
```

Closes the file associated with `handle`, which may have been obtained from `creat`, `open`, `dup`, or `dup2`. Returns 0 on success, else -1 on error and sets `errno` to one of: `EBADF` or `EINTR`.

dup: Synopsis

```
#include <unistd.h>
int dup(int filedes);
```

Duplicates the file handle `filedes`. The duplicated handle will have the same access mode, the same file pointer and same open file or device as `filedes`. Returns the duplicated file handle, else -1 on error and sets `errno` to one of: `EBADF` or `EMFILE`.

dup2: Synopsis

```
#include <unistd.h>
int dup2(int oldfiledes, int newfiledes);
```

Duplicates an old file handle, `oldfiledes` onto an existing new file handle, `newfiledes`. The duplicated handle will have the same access mode, the same file pointer and same open file or device as the old handle. If the file associated with the new file handle is already opened, it will be first closed. Returns 0 on success, else -1 on error and sets `errno` to one of `EBADF` or `EMFILE`.

fork: Synopsis

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Creates a child process, which is a copy of the parent. The child gets a copy of the parent's data space, heap and stack. While `fork` is called once, it will return twice: once to the parent and once to the child. It returns 0 to the child and returns the process `ID` of the child to the parent. Both the child and parent continue executing from the place in the program directly after the call. However, there is no guarantee which process will commence first. On error, it returns -1 and sets `errno` to one of: `EAGAIN` or `ENOMEM`.

getcwd: Synopsis

```
#include <unistd.h>
char *getcwd(char *buf, size_t sz);
```

The `getcwd` function gets the current working directory's absolute path and copies it into the character array `buf`, and the length of `buf` is specified by `sz`. Returns on success `buf`, or `NULL` on error.

getpid: Synopsis

```
#include <sys/types.h>
#include <unistd.h>
int getpid(void);
```

Returns the group ID of the calling process. No error indicated or `errno` designators.

link: Synopsis

```
#include <unistd.h>
int link(const char *oldpath, const char *newpath);
```

Creates a link to an existing file or directory specified by`oldpath`, and give it the name `newpath`. It is an error if the newpath already exists. Note, only a superuser process can create a link to a directory. Returns 0 on success, else -1 on error and sets `errno` to one of: `EINVAL`, `EPERM` or `ESRCH`.

lseek: Synopsis

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int filedes, off_t offset, int whence);
```

Set the file descriptor's, `filedes`, pointer to the new position specified by `offset`, which will be measured relative to `whence`. `whence` can be on of: `SEEK_SET`, `SEEK_END` or `SEEK_CUR`. Returns on success the resulting position in number of bytes from the beginning of the file, else it returns -1 and sets `errno` to one of: `EBADF`, `EINVAL` or `ESPIPE`. Note, `lseek(filedes,0,SEEK_CUR)` will return the current file pointer location.

pause: Synopsis

```
#include <unistd.h>
int pause(void);
```

Causes the program or calling unit to sleep until a signal is received. `Returns` -1.

pipe: Synopsis

```
#include <unistd.h>
int pipe(int filedes[2]);
```

Creates a pipe, with read and write file descriptors stored in filedes[0] and filedes[1] respectively. The output filedes[0] is the input into file filedes[1]. A call to `pipe` is often accompanied by a `fork`, of which the parent process will close the read end of the pipe, while the child process will close the write end. Returns 0 on success, else -1 on error and sets `errno` to one of: `EMFILE` or `ENFILE`.

read: Synopsis

```
#include <unistd.h>
ssize_t read(int filedes, char *buf, size_t count);
```

Attempts to read `count` bytes from `filedes` and will place them in `buf`. One success, returns the number of bytes read, on `end-of-file` zero, otherwise -1 and sets `errno` to one of: `EACCESS`, `EAGAIN`, `EBADF`, `EINTR`, `EISDIR`, or `EFAULT`.

rmdir: Synopsis

```
#include <unistd.h>
int rmdir(const char * path);
```

Removes the directory specified by `path`, which must be empty. Returns zero on success, else -1 on error and `errno` will be set to one of: `EBUSY`, `EFAULT`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, `ENOTEMPTY`, `EPERM` or `EROFS`.

sleep: Synopsis

```
#include <unistd.h>
unsigned int sleep(unsigned int sec);
```

Puts the current process to sleep for `sec` seconds. or until a signal arrives which is not ignored. Returns the amount of time not slept. No error codes used.

unlink: Synopsis

```
#include <unistd.h>
int unlink(const char *fname);
```

Removes the link, `fname` from the filesystem. If it is the last link to the that name and no other process has the file opened, then the file space will also be freed. If there are no other links and any process still has the file opened then the file space will remain in existence until the last file descriptor referring to it is closed. Returns zero on success, else -1 and `errno` is set to one of: `EACCES`, `EFAULT`, `EISDIR`, `ENAMETOOLONG`, `ENOENT`, `ENOMEM`, `ENOTDIR`, `EPERM` , or `EROFS`.

usleep: Synopsis

```
#include <unistd.h>
void usleep(unsigned long usec);
```

The `usleep` function when called waits `usec` microseconds before returning to its caller. Its accuracy depends on the time needed to actually process the call, but as the value of `usec` increase so should its accuracy. 1sec=10^6usec.

write: Synopsis

```
#include <unistd.h>
ssize_t write(int filedes, const char *buf, size_t count);
```

Writes up to `count` bytes to file descriptor `filedes` from from `buf`. Returns the number of bytes written, zero on end-of-file or otherwise -1 and `errno` will be set to one of: `EAGAIN`, `EBADF`, `EFAULT`, `EINTR`, `EINVAL`, `ENOSPC` or `EPIPE`.

4.3 Implementation library support

In this section, those functions that are not part of ISO C or the POSIX.1 standard are presented. These functions will typically be found on most UNIX system.

4.3.1 stdio.h

The EiC header file `<stdio.h>` defines the following extra functions:

pclose: Synopsis

```
#include <stdio.h>
int pclose( FILE *stream);
```

popen: Synopsis

```
#include <stdio.h>
FILE *popen( const char *command, const char *type);
```

4.3.2 dirent.h

The EiC header file `<dirent.h>` defines these extra functions:

seekdir: Synopsis

```
#include <dirent.h>
void seekdir(DIR *dir, off_t offset);
```

telldir: Synopsis

```
#include <dirent.h>
off_t telldir(DIR *dir);
```

Chapter 5

The Advanced Concepts

In this chapter the more esoteric and advanced concepts of EiC will be discussed.

5.1 EiC modules

5.1.1 Building an eic module

There are basically two types of EiC modules. Interpreter'd code modules and modules which get builtin to EiC (compiled code). The simplist modules to construction are interpreter'd modules. In a nutshell intepreter'd modules are related groups of EiC/C functions, which get interpreter'd. Builtin modules are related groups of compiled functions, which have been interfaced to EiC – a process which has now been automated thanks to work by Jean-Bruno Richard.

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. For builtin modules, EiC will need to be clobbered and recompiled after each addition or removal of a builtin module, and this is done from EiC's source code directory:

```
% make clobber
% config/makeconfig
% make install
```

5.1.2 Interpreter'd modules

Adding an interpreter module is simple, just create a directory for the module in the EiC directory 'EiC/module' and place a copy (not a link) of the 'EiC/module/Makefile.empty'

in your module directory and rename it to Makefile.

'Makefile.empty' is a dummy Makefile which is used to prevent the Makefile system from crashing whenever you rebuild EiC. EiC's build process expects a Makefile with various targets in each directory off the 'EiC/module' directory.

Once you have setup the directory and added the Makefile, you can start adding your own code into your module directory. Lets say the new interpreter'd module is called 'foo' and you have code files called 'foo.h' and 'f1.c', these can now be accessed from the EiC prompt by simply entering:

```
EiC > #include foo/foo.h
EiC > #include foo/f1.c
```

Or from any other file, which gets included into an EiC session. This is because the 'EiC/module' directory is by default included in EiC's search list (see pg: 26). Therefore, any include file that belongs to a module only needs to be referenced relative to its module directory.

5.1.3 Module names and assumptions

The name given to a module, *module_name*, is important for several reasons, and particularly so for builtin modules. This is because on startup, EiC will expect to call each builtin-modules initialising function 'module_*module_name*' (see also pg: 157). This call will be automatically setup by EiC's makefile system, using the file 'EiC/module/modules.calls'. Entries into this file will de done via each Makefile installed in each builtin module. The entry for module directory 'XXXX' would look like:

```
#ifndef NO_module_XXXX
module_XXXX();
#endif
```

The file 'EiC/module/modules.calls' is inturn included into EiC's main function.

5.1.4 Building builtin modules

Building a builtin module is considerably more complex than building an interpreter'd module. Basically, a builtin module is a set of interface routines that interface EiC to a library of compiled C code and that also allows compiled C code to make callbacks to EiC. Callbacks are called from builtin code that take as an argument, a pointer to a function; such as qsort does.

There are two basic steps to building a builtin module: (1) construct the interface code files, via EiC's ':gen' command (see also pg: 22) and (2) construct the Makefile, which

is done via a copy and modify approach. It is expected however, that the builder, the developer, of the interface is knowledgeable about the library being interfaced to and will take into consideration which callbacks must be multiplexed and will understand EiC's pointer qualifiers (see § 3.12.5, pg: 66).

5.1.5 Restrictions for builtin functions

At present EiC will not allow a builtin function to be passed a pointer to a function that takes a variable argument list. This is becuase EiC needs to be able to construct the callback code, and to do this it needs to know in advanced what the callback arguments are.

However, EiC's can interface to prototypes of builtin routines that accept as an argument pointers to callback functions which have empty paramater lists: For example, consider the following interpreter'd functions:

```
int f0(void) { return 1;}
int f1(double x) { return x + 0.5;}
int f2(char *s) { return strlen(s););
int f3(int x, int y) { return x + y;}
```

The prototype for a builtin function might be:

```
int fooey( int x, int f() );
```

It is now possible to pass the address of 'f0', 'f1', 'f2' and 'f3' to 'fooey'. The builtin function 'fooey' would then pass the proper arguments to the appropriate callback function, possibly based on the value 'x', which will also be passed to 'fooey'.

5.1.6 Interfacing to a library of C code

For the purpose of discussion some example code will be used. First, the interface between EiC and a library of C code is done via each library's header file(s). Therefore, the following header file, foo.h, will be used:

```
1: /* begin header */
2:
3: extern int GVALUE;
4:
5: extern int foo1(int x, int y);
6: extern int foo2(int z, int (*)(int, int));
7: extern int * foo3(double (*)());
```

This is a simple header file, which is complicated enough to be interesting and is used to demonstrate several principles only. On line 3 there is an external variable that must be shared between the library being interfaced with and the EiC interpreter. Line 5 represents a straight forward function prototype. On line 6 is a little more complicated prototype, as one of the arguments is a pointer to a function which returns an `int` and it accepts two `int` arguments. The prototype on line 7 is the most complicated in the example code, as it takes a pointer to a function that receives an empty parameter list. In C an empty parameter list does not mean that the function accepts zero aguments; i.e. void. On the contrary, it means it can accept a variable number of arguments from zero to N. The only thing that is certain is that arguments cannot be widened – `double` instead of `float`, `int` instead of `short` etc.

Moving to the directory of interest:

```
% cd EiC/module/foo
```

Running EiC and from its command line the following lines are entered:

```
EiC > #include foo.h
EiC > :gen foo.h "foo.c"
```

The ':gen' command takes input from 'foo.h' and creates an output file 'foo.c', which contains the interfaces. Note the output file must be passed to the ':gen' command as a string, and if no output file is given the interfaces will be written to stdout.

The first part of file 'foo.c' includes several header files:

```
#include <stdlib.h>
#include <varargs.h>
#include "eic.h"
#include "foo.h"
```

The varargs.h mechanism is required for passing variable arguments between compiled C code and EiC, and the header file "eic.h" contains macros and prototypes required to access EiC's runtime stack and for generating callbacks .

Next, the interface to 'foo1' generated is given:

```
static val_t eic_foo1(void)
{
        val_t v;

        v.ival = foo1(arg(0,getargs(),int),
                arg(1,getargs(),int));

        return v;
}
```

This is the simplest interface, it essentially just collects the arguments passed on EiC's stack, via the 'arg' facililty defined in the header file "eic.h", passes these values to the function "foo1" and it returns the return value, packaged in the union 'v' to EiC.

This is then followed by the interface to 'foo2':

```
static void * EiC_Cfunc_0 = NULL;
static int  MiddleOne_0(int  x0, int  x1)
{
    setArg(0, EiC_Cfunc_0, int ,x0);
    setArg(1, EiC_Cfunc_0, int ,x1);

    EiC_callBack(EiC_Cfunc_0);
    return EiC_ReturnValue( int );
}

static val_t eic_foo2(void)
{
        val_t v;

        EiC_Cfunc_0 = arg(1,getargs(),ptr_t).p;
        v.ival = foo2(arg(0,getargs(),int),
                MiddleOne_0);

        return v;
}
```

The interface to 'foo2' is more complex and requirs two functions. At compile time, EiC creates the callback code for the function being passed. The callback codes gets substituted for the pointer to the function and a reference to it will be stored in EiC_Cfunc_0 and when the interface routine 'eic_foo2' is called.

Within 'eic_foo2' the compiled function foo2 is called, passing it a pointer to the proxy function 'MiddleOne_0'. The roll of 'MiddleOne_0' is to collect the arguments being passed from 'foo2' for the interpreter'd function pointered to by 'EiC_Cfunc_0', and to return the return value back to 'foo2'. 'MiddleOne_0' uses EiC's setArg facility defined in 'eic.h' for passing values from the machines runtime stack to the EiC interpreter'd function (and it makes no difference if the interpreter'd function being callback is actually another builtin function). The EiC_ReturnValue(type) macro gets the last value stored on EiC's runtime stack and casts it to 'type', and it is this value that gets returned to its caller.

This all occurs seamlessly to the user; for example:

```
EiC > int f(int x, int y) { return x + y;}
EiC > foo2(5,f);
```

On line 7 is the prototype for 'foo3' is given:

```
7: extern int * foo3(double (*)());
```

The 'foo3' function is the most complex funtion in the example code to interface to. This is because 'foo3' receives a pointer to a function, to which inturn accepts a variable number of arguments and it also returns a pointer, adding another degree of complexity to the interface. The default interface generated will be:

```
static void * EiC_Cfunc_1 = NULL;
static double  MiddleOne_1( va_alist )  va_dcl
{
    void Auto_EiC_CallBack(code_t *callback, va_list ap);
    va_list ap; va_start(ap);
    Auto_EiC_CallBack(EiC_Cfunc_1,ap);

    EiC_callBack(EiC_Cfunc_1);
    return EiC_ReturnValue( double );
    va_end(ap);
}

static val_t eic_foo3(void)
{
        val_t v;

        EiC_Cfunc_1 = arg(0,getargs(),ptr_t).p;
        v.p.ep = v.p.sp = v.p.p = foo3(MiddleOne_1);

        return v;
}
```

This interface works essentially the same way as that for 'foo2'. The main differences being that the proxy function 'MiddleOne_1' uses the Unix varargs mechanism for passing variable arguments. However, rather than using the 'setArg' mechanism, the function 'Auto_EiC_callBack' is used to get the variable(s) to be passed to the callback function.

5.1.7 Returning pointers

The next thing to notice in 'eic_foo3' is that it returns a pointer, and therefore, the limits or range for the pointer values must be set. EiC treats all pointers by default as safe. The value of a safe pointer (v.p.p) should always satisfy:

```
v.p.ep >= v.p.p  && v.p.sp <= v.p.p;
```

The end point 'ep' and the start start 'sp' must be set appropriately. The ':gen' command has no insight into the function being interfaced with, so it must take a conservative approach. However, if on the otherhand the developer of the interface new that 'foo3' was going to return a pointer to an area large enough to hold 'N' ints for example, then the following change would be appropriate:

```
v.p.ep = v.p.sp = v.p.p = foo3(MiddleOne_1);
```

to

```
v.p.sp = v.p.p = foo3(MiddleOne_1);
v.p.ep = (char*)v.p.p + N * sizeof(int);
```

For further details on EiC pointers see: § 3.12.5, pg: 66.

5.1.8 Initialising the module

The last part of the file generaterd by ':gen' contains the function 'module_*module_name*', where for this example the *module_name* is 'foo':

```
void module_foo()
{
        EiC_parseString("int GVALUE @ 0x%p;", &GVALUE);
        add_builtinfunc("foo1",eic_foo1);
        add_builtinfunc("foo2",eic_foo2);
        add_builtinfunc("foo3",eic_foo3);
}
```

First the initialising function 'module_foo' setups the shared variables using EiC's address operator, as proposed by Eugene Brooks III. Next, each builtin function is added to EiC's lookup tables. The first variable passed to the 'add_builtinfunc' function is the name of the function that will be seen by the EiC interpreter, and the second argument is the function that will be actually called.

Also, the 'module_foo' function is the initialising function for the module 'foo' (see also pg: 152). This is because the module directory name is also 'foo'. Each builtin modules initialising function will be called automatically on EiC startup. If there are other header files in the directory 'foo' that must be intialised, then it is expected that developer will insert calls to these other files within the initialising function module. Although this feature will most likely be automated in a future release of EiC.

5.1.9 Multiplexed interfacing

There is still the problem of multiplexing that must be addressed; that is, interfacing to builtin functions that allow the call of different functions according to predefined signals. At this stage, it is upto the developer of the interface to decide which functions must be mutliplexed and the level of multiplexing required. For example, consider the problem of interfacing to a menu function that will call different functions depending on which menu item is selected. As an example, the interface to 'foo2' will be modified for 3 levels of multiplexing; that is, at any given instant it may make callbacks on anyone of 3 functions:

```
#define ML_0  3
static int cbs_0 = 0;

static void *EiC_Cfunc_0[ML_0];

static int  MiddleOne_0(int x, int  x0, int  x1)
{
    setArg(0, EiC_Cfunc_0[x], int ,x0);
    setArg(1, EiC_Cfunc_0[x], int ,x1);

    EiC_callBack(EiC_Cfunc_0[x]);
    return EiC_ReturnValue( int );
}

static int MiddleOne_0a(int x, int y) { return MiddleOne_0(0, x,y); }
static int MiddleOne_0b(int x, int y) { return MiddleOne_0(1, x,y); }
static int MiddleOne_0c(int x, int y) { return MiddleOne_0(2, x,y); }

static void (*tabFunc_0[])() = {
        MiddleOne_0a,
        MiddleOne_0b,
        MiddleOne_0c,
};

static val_t eic_foo2(void)
{
        val_t v;

        if(cbs_0 == ML_0) {
          fprintf(stderr,"EiC : too many callbacks for foo2\n");
          return v;
        }
```

```
        EiC_Cfunc_0[cbs_0] = arg(1,getargs(),ptr_t).p;
        v.ival = foo2(arg(0,getargs(),int),
                tabFunc_0[cbs_0]);

        cb_0++;

        return v;
}
```

The variable 'EiC_Cfunc_0' is now defined as an array of pointers and the 'MiddleOne_0' function has been interfaced to via an array of pointers to functions, each of which will pass the index to the selected callback code stored in the 'EiC_Cfunc_0' array, as well as the variables passed from 'foo2'. This is done via the use of the function array 'tabFunc_0'. On entry into 'eic_foo2' a pointer to the callback function is stored in 'EiC_Cfunc_0' and a pointer to a function stored in the function pointer array 'tabFunc_0' is passed to 'foo2'. In a interactive environment it might also be appropriate to first search the array 'EiC_Cfunc_0', to see if the incoming pointer is already stored in 'EiC_Cfunc_0' and to be able to reset 'cb_0' back to zero.

5.1.10 Builtin-module's makefiles

The next thing that is required when building a builtin module, is a makefile. To construct the builtin-module's Makefile, copy the 'EiC/module/Makefile.builtin' to the module directory and rename it to 'Makefile'. Next only the following variables within the 'Makefile' will need to set:

```
MODULE =
LINK_LIBS =
libSRCS =
libOBJS =
```

The variable 'MODULE' is used to record the module name. The 'LINK_LIBS' variable should contain the names of the libraries being linked to and any auxially libraries required. Remember a builtin module is just an interface to a C library. The 'libSRCS' variable will contain then names of the C files that where generated by EiC's ':gen' command or any other C code. The 'libOBJS' variable will be assigned the objects to be linked into EiC. For our example, the above lines would be changed to:

```
MODULE = foo
LINK_LIBS = -L/path_2_foo_library -lfoo
```

```
libSRCS =  foo.c
libOBJS = $(LIB)(foo.o)
```

The 'LIB' variable is a predefined to EiC's library libeic. If a second interface file was needed, say fooey, then libSRCS and libOBJS would change to:

```
libSRCS =  foo.c fooey.c
libOBJS = $(LIB)(foo.o) $(LIB)(fooey.o)
```

Appendix A

Syntax of the EiC language

In this section the grammar for the C part of the EiC language is given. For the purposes of comparison, the syntax for the ISO C language from (Kernighan and Ritchie, 1988) is given along side EiC's. Any part of ISO's C grammar that is underlined is currently not supported by EiC. Any part of EiC's C grammar that is underlined in not supported by ISO C.

A.1 Syntax Notation

The following notation is used:

1. Non terminals are given in *italics*.
2. Terminals are given in `typewriter` font.
3. An item surrounded by upright square brackets [ ] denotes that the item is optional. If more than one item is present within a pair of upright square brackets and the items are separated by commas then the square brackets are used for grouping purposes. However, if the first token in a set of tokens is a circumflex (^), then the expression will match any token except those in the set. For example [`int`, `char`], specifies that the input token can be an `int` or `char`, while [^ `(`] specifies the input token can be anything but the left parenthesis. Note however, that square brackets in `typewriter` font, `[ ]`, are terminals.
4. The left side of a production is on a line by itself followed by a colon.
5. The right side, or the definition, of a production, and each definition for each production rule, will be placed on a line by itself and below the left side. For example:

left-side:
    definition$_1$
    definition$_2$
    ⋮
    definition$_n$

6. The `one of` terminology is also used to specify a list of alternatives:

   store-class: `one of`
   `auto register static`
   `extern typedef`

7. a^+ is used to denote a sequence of one or more a's, where a can be anything, including [ ]; in which case the square brackets are used for grouping purposes.

8. a^* is used to denote a sequence of zero or more a's where a can be anything, including [ ]; in which case the square brackets are used for grouping purposes.

9. To restate, ISO C productions, or parts thereof, that have been underlined, are not included in EiC's C grammar. EiC C productions, or parts thereof, that have been underlined, are not included in ISO's C grammar.

EiC LL(2) GRAMMAR	ISO C LR GRAMMAR
ext-decl: *decl-spec f-ext-decl* *f-ext-decl:* *decl ff-ext-decl* ; *ff-ext-decl:* *comp-stmt* = *initialiser fff-ext-decl* *fff-ext-decl* *fff-ext-decl:* ; , *init-decl-list* ;	*ext-decl:* *func-def* *declaration* *func-def:* *decl-spec decl* [*decl-list*] *comp-stmt* *decl* [*decl-list*] *comp-stmt* *declaration:* *decl-spec* [*init-decl-list*] ;
decl-spec: *store-class* [*decl-spec*] *type-spec* [*decl-spec*] *type-qual* [*decl-spec*]	*decl-spec:* *store-class* [*decl-spec*] *type-spec* [*decl-spec*] *type-qual* [*decl-spec*]

store-class: `one of`
`auto register static`
`extern typedef`

type-qual: `one of`
`const volatile`
type-spec: `one of`
`void char short int`
`long float double`
`signed unsigned` *struct-or-union*
typedef-name enum-spec
type-name:
spec-qual-list [*abs-decl*]
typedef-name:
id

enum-spec:
`enum` *f-enum-spec*
f-enum-spec:
{ *enum-list* }
id [{ *enum-list* }]
enum-list:
enumerator [`,` *enumerator*]*
enumerator:
id [`=` *const-expr*]

init-decl-list:
init-decl[`,` *init-decl-list*]*
init-decl:
decl [`=` *initialiser*]
decl:
[*pointer*] *dir-decl*
pointer:
[`*` [*pointer-qual-list*]*] +
pointer-qual-list:
type-qual-list [*pointer-qual*]
pointer-qual [*type-qual-list*]
pointer-qual: `one of`
`safe unsafe`

store-class: `one of`
`auto register static`
`extern typedef`

type-qual: `one of`
`const volatile`
type-spec: `one of`
`void char short int`
`long float double`
`signed unsigned` *struct-or-union*
typedef-name enum-spec
type-name:
spec-qual-list [*abs-decl*]
typedef-name:
id

enum-spec:
`enum` [*id*] {*enum-list*}
`enum` *id*
enum-list:
enumerator
enum-list `,` *enumerator*
enumerator:
id
id `=` *const-expr*

init-decl-list:
init-decl
init-decl-list `,` *init-decl*
init-decl:
decl
decl `=` *initialiser*
decl:
[*pointer*] *dir-decl*
pointer:
`*` [*type-qual-list*]
`*` [*type-qual-list*] *pointer*
type-qual-list:
type-qual
type-qual-list type-qual

```
abs-decl:
    pointer f-abs-decl
    dir-abs-decl
f-abs-decl:
    dir-abs-decl
    null
```

```
abs-decl:
    pointer
    [pointer] dir-abs-decl
```

```
dir-abs-decl:
    ( f1-dir-ab f2-dir-abs
    array-decl f2-dir-abs
f1-dir-abs:
    abs-decl )
    ff-dir-decl
f2-dir-abs:
    array-decl f2-dir-abs
    ( ff-dir-decl f2-dir-abs
    null
```

```
dir-abs-decl:
    ( abs-decl )
    [dir-abs-decl][ [const-expr] ]
    [dir-abs-decl] ( [par-type-list] )
```

```
dir-decl:
    id f-dir-decl
    ( decl ) f-dir-decl
f-dir-decl:
    array-decl f-dir-decl
    ( ff-dir-decl f-dir-decl
    null
ff-dir-decl:
    [parm-type-list] )
```

```
dir-decl:
    id
    ( decl )
    dir-decl [ [const-expr] ]
    dir-decl ( parm-type-list )
    dir-decl ( [ident-list] )
```

```
parm-type-list:
    parm-decl f-parm-type-list
f-parm-type-list:
    , ff-parm-type-list
    null
ff-parm-type-list:
    ...
    parm-type-list
```

```
parm-type-list:
    parm-list
    parm-list , ...
parm-list:
    parm-decl
    parm-list , parm-decl
```

parm-decl:
    decl-spec f-parm-decl
f-parm-decl:
    pointer ff-parm-decl
    ff-parm-decl
ff-parm-decl:
    `(` *f-parm-decl* `)`
    id f-dir-decl
    array-decl f-dir-decl
    null
array-decl:
    `[` [*const-expr*] `]`

parm-decl:
    decl-spec decl
    decl-spec [*abs-decl*]

st-un-spec:
    st-un f-st-un-spec
f-st-un-spec:
    id [{ *s-decl-list* }]
    { *s-decl-list* }

st-un-spec:
    st-un [*id*] { *s-decl-list* }
    st-un id

st-un: `one of`
    `struct union`
s-decl-list:
    [*st-decl*]+
st-decl:
    spec-qual-list spec-declor-list `;`
spec-qual-list:
    [*type-spec*]+
    [*type-qual*]+

st-un: `one of`
    `struct union`
s-decl-list:
    st-decl
    s-decl-list st-decl
st-decl:
    spec-qual-list spec-declor-list `;`
spec-qual-list:
    type-spec [*spec-qual-list*]
    type-qual [*spec-qual-list*]

spec-declor-list:
    st-declor [`,`*spec-declor-list*]*

spec-declor-list:
    st-declor
    spec-declor-list `,` *st-declor*

st-declor:
    decl f-st-declor
    `:` *const-expr*
f-st-declor:
    `:` *const-expr*
    `null`

st-declor:
    decl
    [*decl*] `:` *const-expr*

stmt:
label-stmt
expr-stmt
comp-stmt
select-stmt
iter-stmt
jump-stmt

label-stmt:
label `:` *stmt*
`case` *const-expr* `:` *stmt*
`default` `:` *stmt*

expr-stmt:
expr `;`
comp-stmt:
`{` [*decl-list*] [*stmt-list*] `}`
stmt-list:
[*stmt*]+

select-stmt:
`if` `(` *expr* `)` *stmt* [`else` *stmt*]
`switch` `(` *expr* `)` *stmt*

iter-statement:
`while` `(` *expr* `)` *stmt*
`do` *stmt* `while` `(` *expr* `)` `;`
`for` `(` [*expr*] `;` [*expr*] `;` [*expr*]) *stmt*

jump-stmt:
`goto` *id* `;`
`continue` `;`
`break` `;`
`return` [*expr*] `;`

expr:
assign-expr [`,` *assign-expr*]

assign-expr:
cond-expr[*assignment-op assign-expr*]

stmt:
label-stmt
expr-stmt
comp-stmt
select-stmt
iter-stmt
jump-stmt

label-stmt:
label `:` *stmt*
`case` *const-expr* `:` *stmt*
`default` `:` *stmt*

expr-stmt:
expr `;`
comp-stmt:
`{` [*decl-list*] [*stmt-list*] `}`
stmt-list:
[*stmt*]+

select-stmt:
`if` `(` *expr* `)` *stmt*
`if` `(` *expr* `)` *stmt* `else` *stmt*
`switch` `(` *expr* `)` *stmt*

iter-statement:
`while` `(` *expr* `)` *stmt*
`do` *stmt* `while` `(` *expr* `)` `;`
`for` `(` [*expr*] `;` [*expr*] `;` [*expr*])*stmt*

jump-stmt:
`goto` *id* `;`
`continue` `;`
`break` `;`
`return` [*expr*] `;`

expr:
assign-expr
expr `,` *assign-expr*

assign-expr:
cond-expr
unary-expr assignment-op assign-expr

assignment-op:
    `=`
    `[*, /,%, +, -, >>, <<, &,^,|] =`

cond-expr:
    log-or-expr `[?` *expr* `:` *cond-expr*`]`

log-or-expr:
    log-and-expr `[||` *log-and-expr*`]*`

log-and-expr:
    inc-or-expr `[&&` *inc-or-expr*`]*`

inc-or-expr:
    xor-expr `[|` *xor-expr*`]*`

xor-expr:
    and-expr `[^` *and-expr*`]*`

and-expr:
    equal-expr `[&` *equal-expr*`]*`

equal-expr:
    rel-expr `[[==` , `!=]` *rel-expr*`]*`

rel-expr:
    shift-expr `[[>,<,<=,>=]` *shift-expr*`]*`

assignment-op:
    `=`
    `[*, /,%, +, -, >>, <<, &,^,|] =`

cond-expr:
    log-or-expr
    log-or-expr `?` *expr* `:` *cond-expr*

log-or-expr:
    log-and-expr
    log-or-expr `||` *log-and-expr*

log-and-expr:
    inc-or-expr
    log-and-expr `&&` *inc-or-expr*

inc-or-expr:
    xor-expr
    inc-or-expr `|` *xor-expr*

xor-expr:
    and-expr
    xor-expr `^` *and-expr*

and-expr:
    equal-expr
    and-expr `&` *equal-expr*

equal-expr:
    rel-expr
    equal-expr `==` *rel-expr*
    equal-expr `!=` *rel-expr*

rel-expr:
    shift-expr
    rel-expr `<` *shift-expr*
    rel-expr `>` *shift-expr*
    rel-expr `<=` *shift-expr*
    rel-expr `>=` *shift-expr*

shift-expr:
    add-expr [[<<,>>] *add-expr*]*

shift-expr:
    add-expr
    shift-expr << *add-expr*
    shift-expr >> *add-expr*

add-expr:
    mult-expr [[+,-] *mult-expr*]*

add-expr:
    mult-expr
    add-expr + *mult-expr*
    add-expr - *mult-expr*

mult-expr:
    cast-expr [[*,/,%] *cast-expr*]*

mult-expr:
    cast-expr
    mult-expr * *cast-expr*
    mult-expr / *cast-expr*
    mult-expr % *cast-expr*

cast-expr:
    [^ (] *unary-expr*
    (*f-cast-expr*
f-cast-expr:
    type-name) *cast-expr*
    expr) *r-postfix-expr*

cast-expr:
    unary-expr
    (*type-name*) *cast-expr*

unary-expr:
    postfix-expr
    [++,--] *unary-expr*
    [&, *, +, -, ~, !] *cast-expr*
    sizeof *sizeof-ext*
sizeof-ext:
    [^ (] *unary-expr*
    ([*type-name, unary-expr*])

unary-expr:
    postfix-expr
    [++,--] *unary-expr*
    [&, *, +, -, ~, !] *cast-expr*
    sizeof [(*type-name*), *unary-expr*]

postfix-expr:
    primary-expr r-postfix-expr
r-postfix-expr:
    [*expr*] *r-postfix-expr*
    ([*arg-expr-list*]) *r-postfix-expr*
    . *id r-postfix-expr*
    -> *id r-postfix-expr*
    ++ *r-postfix-expr*
    -- *r-postfix-expr*
    null

postfix-expr:
    primary-expr
    postfix-expr [*expr*]
    postfix-expr ([*arg-expr-list*])
    postfix-expr . *id*
    postfix-expr -> *id*
    postfix-expr ++
    postfix-expr --

arg-expr-list:
    assign-expr [, *assign-expr*]

primary-expr:
    id
    constant
    string
    (*expr*)

const-expr:
    cond-expr

constant:
    int-const
    float-const
    char-const
    enum-const
    typename

arg-expr-list:
    assign-expr
    arg-expr-list , *assign-expr*

primary-expr:
    id
    constant
    string
    (*expr*)

const-expr:
    cond-expr

constant:
    int-const
    float-const
    char-const
    enum-const

Bibliography

Budd, T. (1987). *A Little Smalltalk*. Addison-Wesley, Reading, MA, USA.

Fraser, C. W. and Hanson, D. R. (1995). *A retargetable C compiler: design and implemenation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ 07632, USA, second edition.

Pemberton, S. and Daniels, M. (1982). *Pascal Implementation - the P4 Compiler*. Eillis Horwood, Chichester.

Stevens, W. R. (1992). *Advanced Programming in the UNIX Environment*. Addison-Wesley, Reading, MA, USA.

Zlotnick, F. (1991). *The POSIX.1 Standard: A Programmer's Guide*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA.

On line documentation and C sources

- Steve Summit's Introductory C course:
  `http://www.eskimo.com/~scs/cclass/cclass.html`
- Martin Leslie's Online C Language Reference:
  `http://users.southeast.net/~garyg/C_ref/C/c.html`
- Ross Richardson's The C Standard Library:
  `http://www.infosys.utas.edu.au/info/documentation/C/CStdLib.html`
- comp.lang.c Frequently Asked Questions:
  `http://www.eskimo.com/~scs/C-faq/top.html`
- Other Sources, C, Related Languages, Programming Languages:
  `http://www.lysator.liu.se/c/c-www.html`

Index