Header files

When the C language was young, header files were required to define structures and occasionally to specify that a function did something out of the ordinary like taking a `double` parameter or returning a `float` result. Then ANSI C and POSIX came along and changed all that.

Header files seem a relatively simple idea, but in fact they can be a major source of annoyance in porting. In particular:

- ANSI and POSIX.1 have added a certain structure to the usage of header files, but there are still many old-fashioned headers out there.
- ANSI and POSIX.1 have also placed more stringent requirements on data types used in header files. This can cause conflicts with older systems, especially if the author has commited the sin of trying to out-guess the header files.
- C++ has special requirements of header files. If your header files don't fulfil these requirements, the GNU *protoize* program can usually fix them.
- There is still no complete agreement on the names of header files, or in which directories they should be placed. In particular, System V.3 and System V.4 frequently disagree as to whether a header file should be in */usr/include* or in */usr/include/sys*.

ANSI C, POSIX.1, and header files

ANSI C and POSIX.1 have had a far-reaching effect on the structure of system header files. We'll look at the changes in the C language in more detail in Chapter 20, *Compilers*. The following points are relevant to the use of header files:

- ANSI C prefers to have an ANSI-style prototype for every function call it encounters. If it doesn't find one, it can't check the function call semantics as thoroughly, and it may issue a warning. It's a good idea to enable all such warnings, but this kind of message makes it difficult to recognize the real errors hiding behind the warnings. In C++, the rules are even stricter: if you don't have a prototype, it's an error and your source file doesn't compile.

- To do a complete job of error checking, ANSI C requires the prototype in the new, embedded form:

  ```
  int foo (char *zot, int glarp);
  ```

  and not

  ```
  int foo (zot, glarp);
  char *zot;
  ```

  Old C compilers don't understand this new kind of prototype.

- Header files usually contain many definitions that are not part of POSIX.1. A mechanism is needed to disable these definitions if you are compiling a program intended to be POSIX.1 compatible.*

The result of these requirements is spaghetti header files: you frequently see things like this excerpt from the header file *stdio.h* in 4.4BSD:

```
/*
 * Functions defined in ANSI C standard.
 */
__BEGIN_DECLS
void     clearerr __P((FILE *));
int      fclose __P((FILE *));

#if !defined(_ANSI_SOURCE) && !defined(_POSIX_SOURCE)
extern int sys_nerr;          /* perror(3) external variables */
extern __const char *__const sys_errlist[];
#endif
void     perror __P((const char *));

__END_DECLS

/*
 * Functions defined in POSIX 1003.1.
 */
#ifndef _ANSI_SOURCE
#define L_cuserid       9     /* size for cuserid(); UT_NAMESIZE + 1 */
#define L_ctermid       1024  /* size for ctermid(); PATH_MAX */

__BEGIN_DECLS
char    *ctermid __P((char *));

__END_DECLS
#endif /* not ANSI */

/*
 * Routines that are purely local.
 */
```

* Writing your programs to conform to POSIX.1 may be a good idea if you want them to run on as many platforms as possible. On the other hand, it may also be a bad idea: POSIX.1 has very rudimentary facilities in some areas. You may find it more confining than is good for your program.

```
#if !defined (_ANSI_SOURCE) && !defined(_POSIX_SOURCE)
__BEGIN_DECLS
char    *fgetln __P((FILE *, size_t *));

__END_DECLS
```

Well, it *does* look vaguely like C, but this kind of header file scares most people off. A number of conflicts have led to this kind of code:

- The ANSI C library and POSIX.1 carefully define a subset of the total available functionality. If you want to abide strictly to the standards, any extension must be flagged as an error, even if it would work.
- The C++ language has a different syntax from C, but both languages share a common set of header files.

These solutions have caused new problems, which we'll examine in this chapter.

ANSI and POSIX.1 restrictions

Most current UNIX implementations do not conform completely with POSIX.1 and ANSI C, and every implementation offers a number of features that are not part of either standard. A program that conforms with the standards must not use these features. You can specify that you wish your program to be compliant with the standards by defining the preprocessor variables `_ANSI_SOURCE` or `_POSIX_SOURCE`, which maximizes the portability of the code. It does this by preventing the inclusion of certain definitions. In our example, the array `sys_errlist`, (see Chapter 18, *Function libraries*, page 298), is not part of POSIX.1 or ANSI, so the definition is not included if either preprocessor variable is set. If we refer to `sys_errlist` anyway, the compiler signifies an error, since the array hasn't been declared. Similarly, `L_cuserid` is defined in POSIX.1 but not in ANSI C, so it is defined only when `_POSIX_SOURCE` is defined and `_ANSI_SOURCE` is not defined.

Declarations for C++

C++ has additional requirements of symbol naming: *function overloading* allows different functions to have the same name. Assemblers don't think this is funny at all, and neither do linkers, so the names need to be changed to be unique. In addition, the names need to somehow reflect the class to which they belong, the kind of parameters that the function takes and the kind of value it returns. This is done by a technique called *function name encoding*, usually called *function name mangling*. The parameter and return value type information is appended to the function name according to a predetermined rule. To illustrate this, let's look at a simple function declaration:

```
double Internal::sense (int a, unsigned char *text, Internal &p, ...);
```

- First, two underscores are appended to the name of the function. With the initial underscore we get for the assembler, the name is now `_sense__`.

- Then the class name, `Internal` is added. Since the length of the name needs to be specified, this is put in first: `_sense__8Internal`.
- Next, the parameters are encoded. Simple types like int and char are abbreviated to a single character (in this case, `i` and `c`. If they have modifiers like `unsigned`, these, too, are encoded and precede the type information. In this case, we get just plain `i` for the int parameter, and `PUc` (a `P`ointer to `U`nsigned `c`haracters for the second parameter: `_sense__8InternaliPUc`.
- Class or structure references again can't be coded ahead of time, so again the length of the name and the name itself is used. In this case, we have a reference, so the letter `R` is placed in front of the name: `_sense__8InternaliPUcR8Internal`.
- Finally, the ellipses are specified with the letter `e`: `_sense__8InternaliPUcR8Internale`.

For more details on function name mangling, see *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup.

This difference in naming is a problem when a C++ program really needs to call a function written in C. The name in the object file is not mangled, and so the C++ compiler must not output a reference to a mangled name. Theoretically, there could be other differences between C++ calls and C calls that the compiler also needs to take into account. You can't just assume that a function written in another language adheres to the same conventions, so you have to tell it when a called function is written according to C conventions rather than according to C++ conventions.

This is done with the following elegant construct:

```
extern "C"
  {
  char  *ctermid (char *);
  };
```

In ANSI C, the same declaration would be

```
char  *ctermid (char *);
```

and in K&R C it would be

```
char  *ctermid ();
```

It would be a pain to have a separate set of header files for each version. Instead, the implementors defined preprocessor variables which evaluate to language constructs for certain places:

- `__BEGIN_DECLS` is defined as `extern "C" {` for C++ and nothing otherwise.
- `__END_DECLS` is defined as `};` for C++ and nothing otherwise.
- `__P(foo)` is defined as `foo` for C++ and ANSI C, and nothing otherwise. This is the reason why the arguments to `__P()` are enclosed in double parentheses: the outside level of parentheses gets stripped by the preprocessor.

In this implementation, *sys/cdefs.h* defines these preprocessor variables. What happens if *sys/cdefs.h* isn't included before *stdio.h*? Lots of error messages. So one of the first lines in *stdio.h* is `#include <sys/cdefs.h>`. This is not the only place that *sys/cdefs.h* is included: in this particular implementation, from 4.4BSD, it is included from *assert.h*, *db.h*, *dirent.h*, *err.h*, *fnmatch.h*, *fstab.h*, *fts.h*, *glob.h*, *grp.h*, *kvm.h*, *locale.h*, *math.h*, *netdb.h*, *nlist.h*, *pwd.h*, *regex.h*, *regexp.h*, *resolv.h*, *runetype.h*, *setjmp.h*, *signal.h*, *stdio.h*, *stdlib.h*, *string.h*, *time.h*, *ttyent.h*, *unistd.h*, *utime.h* and *vis.h*. This places an additional load on the compiler, which reads in a 100 line definition file multiple times. It also creates the possibility for compiler errors. *sys/cdefs.h* defines a preprocessor variable `_CDEFS_H_` in order to avoid this problem: after the obligatory UCB copyright notice, it starts with

```
#ifndef _CDEFS_H_
#define _CDEFS_H_

#if defined(__cplusplus)
#define __BEGIN_DECLS   extern "C" {
#define __END_DECLS     };
#else
#define __BEGIN_DECLS
#define __END_DECLS
#endif
```

This is a common technique introduced by ANSI C: the preprocessor only processes the body of the header file the first time. After that, the preprocessor variable `_CDEFS_H_` is defined, and the body will not be processed again.

There are a couple of things to note about this method:

- There are no hard and fast rules about the naming and definition of these auxiliary variables. The result is that not all header files use this technique. For example, in FreeBSD 1.1, the header file *machine/limits.h* defines a preprocessor variable `_MACHINE_LIMITS_H` and only interprets the body of the file if this preprocessor variable was not set on entry. BSD/OS 1.1, on the other hand, does not. The same header file is present, and the text is almost identical, but there is nothing to stop you from including and interpreting *machine/limits.h* multiple times. The result can be that a package that compiles just fine under FreeBSD may fail to compile under BSD/OS.
- The ANSI standard defines numerous standard preprocessor variables to ensure that header files are interpreted only the first time they are included. The variables all start with a leading _, and the second character is either another _ or an upper-case letter. It's a good idea to avoid using such symbols in your sources.
- We could save including *sys/cdefs.h* multiple times by checking `_CDEFS_H_` before including it. Unfortunately, this would establish an undesireable relationship between the two files: if for some reason it becomes necessary to change the name of the preprocessor variable, or perhaps to give it different semantics (like giving it different values at different times, instead of just being defined), you have to go through all the header files that refer to the preprocessor variable and modify them.

ANSI header files

The ANSI C language definition, also called *Standard C*, was the first to attempt some kind of standardization of header files. As far as it goes, it works well, but unfortunately it covers only a comparatively small number of header files. In ANSI C,

- The only header files you should need to include are *assert.h*, *ctype.h*, *errno.h*, *float.h*, *limits.h*, *locale.h*, *math.h*, *setjmp.h*, *signal.h*, *stdarg.h*, *stddef.h*, *stdio.h*, *stdlib.h*, *string.h* and *time.h*.
- You may include headers in any order.
- You may include any header more than once.
- Header files do not depend on other header files.
- Header files do not include other header files.

If you can get by with just the ANSI header files, you won't have much trouble. Unfortunately, real-life programs usually require headers that aren't covered by the ANSI standard.

Type information

A large number of system and library calls return information which can be represented in a single machine word. The machine word of the PDP-11, on which the Seventh Edition ran, was only 16 bits wide, and in some cases you had to squeeze the value to get it in the word. For example, the Seventh Edition file system represented an inode number in an `int`, so each file system could have only 65536 inodes. When 32-bit machines were introduced, people quickly took the opportunity to extend the length of these fields, and modern file systems such as *ufs* or *vxfs* have 32 bit inode numbers.

These changes were an advantage, but they bore a danger with them: nowadays, you can't be sure how long an inode number is. Current systems really do have different sized fields for inode numbers, and this presents a portability problem. Inodes aren't the only thing that has changed: consider the following structure definition, which contains information returned by system calls:

```
struct process_info
  {
  long pid;                   /* process number */
  long start_time;            /* time process was started, from time () */
  long owner;                 /* user ID of owner */
  long log_file;              /* file number of log file */
  long log_file_pos;          /* current position in log file */
  short file_permissions;     /* default umask */
  short log_file_major;       /* major device number for log file */
  short log_file_minor;       /* minor device number */
  short inode;                /* inode number of log file */
  }
```

On most modern systems, the `long`s take up 32 bits and the `short`s take up 16 bits. Because

of alignment constraints, we put the longest data types at the front and the shortest at the end (see Chapter 11, *Hardware dependencies*, page 158 for more details). And for older systems, these fields are perfectly adequate. But what happens if we port a program containing this structure to a 64 bit machine running System V.4 and *vxfs*? We've already seen that the inode numbers are now 32 bits long, and System V.4 major and minor device numbers also take up more space. If you port this package to 4.4BSD, the field `log_file_pos` needs to be 64 bits long.

Clearly, it's an oversimplification to assume that any particular kind of value maps to a `short` or a `long`. The correct way to do this is to define a type that describes the value. In modern C, the structure above becomes:

```
struct process_info
  {
  pid_t   pid;                /* process number */
  time_t  start_time;         /* time process was started, from time () */
  uid_t   owner;              /* user ID of owner */
  long    log_file;           /* file number of log file */
  pos_t   log_file_pos;       /* current position in log file */
  mode_t  file_permissions;   /* default umask */
  short   log_file_major;     /* major device number for log file */
  short   log_file_minor;     /* minor device number */
  inode_t inode;              /* inode number of log file */
  }
```

It's important to remember that these type definitions are all in the mind of the compiler, and that they are defined in a header file, which is usually called *sys/types.h*: the system handles them as integers of appropriate length. If you define them in this manner, you give the compiler an opportunity to catch mistakes and generate more reliable code. Check your man pages for the types of the arguments on your system if you run into trouble. In addition, Appendix A, *Comparative reference to UNIX data types*, contains an overview of the more common types used in UNIX systems.

Classes of header files

If you look at the directory hierarchy */usr/include*, you may be astounded by the sheer number of header files, over 400 of them on a typical UNIX system. Fortunately, many of them are in subdirectories, and you usually won't have to worry about them, except for one subdirectory: */usr/include/sys*.

/usr/include/sys

In early versions of UNIX, this directory contained the header files used for compiling the kernel. Nowadays, this directory is intended to contain header files that relate to the UNIX implementation, though the usage varies considerably. You will frequently find files that directly include files from */usr/include/sys*. In fact, it may come as a surprise that this is not supposed to be necessary. Often you will also see code like

```
#ifdef USG                      /* System V */
#include <sys/err.h>
#else                           /* non-System V system */
#include <err.h>
#endif
```

This simplified example shows what you need to do because System V keeps the header file *err.h* in */usr/include/sys*, whereas other flavours keep it in */usr/include*. In order to include the file correctly, the source code needs to know what kind of system it is running on. If it guesses wrong (for example, if `USG` is not defined when it should be) or if the author of the package didn't allow for System V, either out of ignorance, or because the package has never been compiled on System V before, then the compilation will fail with a message about missing header files.

Frequently, the decisions made by the kind of code in the last example are incorrect. Some header files in System V have changed between System V.3 and System V.4. If, for example, you port a program written for System V.4 to System V.3, you may find things like

```
#include <wait.h>
```

This will fail in most versions of System V.3, because there is no header file */usr/include/wait.h*; the file is called */usr/include/sys/wait.h*. There are a couple of things you could do here:

- You could start the compiler with a supplementary `-I/usr/include/sys`, which will cause it to search */usr/include/sys* for files specified without any pathname component. The problem with this approach is that you need to do it for every package that runs into this problem.
- You could consider doing what System V.4 does in many cases: create a file called */usr/include/wait.h* that contains just an obligatory copyright notice and an `#include` directive enclosed in `#ifdef`s:

```
/* THIS IS PUBLISHED NON-PROPRIETARY SOURCE CODE OF O'REILLY  */
/* AND ASSOCIATES Inc.                                        */
/* The copyright notice above does not evidence any actual or */
/* intended restriction on the use of this code.              */
#ifndef _WAIT_H
#define _WAIT_H
#include <sys/wait.h>
#endif
```

Problems with header files

It's fair to say that no system is supplied with completely correct system header files. Your system header files will probably suffer from at least one of the following problems:

- "Incorrect" naming. The header files contain the definitions you need, but they are not in the place you would expect.
- Incomplete definitions. Function prototypes or definitions of structures and constants are missing.

- Incompatible definitions. The definitions are there, but they don't match your compiler. This is particularly often the case with C++ on systems that don't have a native C++ compiler. The *gcc* utility program *protoize*, which is run when installing *gcc*, is supposed to take care of these differences, and it may be of use even if you choose not to install *gcc*.
- Incorrect `#ifdefs`. For example, the file may define certain functions only if `_POSIX_SOURCE` is defined, even though `_POSIX_SOURCE` is intended to restrict functionality, not to enable it. The System V.4.2 version *math.h* surrounds `M_PI` (the constant pi) with

  ```
  #if (__STDC__ && !defined(_POSIX_SOURCE)) || defined(_XOPEN_SOURCE)
  ```

  In other words, if you include *math.h* without defining `__STDC__` (ANSI C) or `_XOPEN_SOURCE` (X Open compliant), `M_PI` will not be defined.
- The header files may contain syntax errors that the native compiler does not notice, but which cause other compilers to refuse them. For example, some versions of XENIX `curses.h` contain the lines:

  ```
  #ifdef M_TERMCAP
  #  include <tcap.h>          /* Use: cc -DM_TERMCAP ...  -lcurses -ltermlib */
  #else
  # ifdef M_TERMINFO
  #  include <tinfo.h>         /* Use: cc -DM_TERMINFO ...  -ltinfo [-lx] */
  # else
     ERROR -- Either "M_TERMCAP" or "M_TERMINFO" must be #define'd.
  # endif
  #endif
  ```

  This does not cause problems for the XENIX C compiler, but *gcc*, for one, complains about the unterminated character constant starting with `define'd`.
- The header files may be "missing". In the course of time, header files have come and gone, and the definitions have been moved to other places. In particular, the definitions that used to be in *strings.h* have been moved to *string.h* (and changed somewhat on the way), and *termio.h* has become *termios.h* (see Chapter 15, *Terminal drivers*, page 241 for more details).

The solutions to these problems are many and varied. They usually leave you feeling dissatisfied:

- Fix the system header files. This sounds like heresy, but if you have established beyond any reasonable doubt that the header file is to blame, this is about all you can do, assuming you can convince your system administrator that it is necessary. If you do choose this way, be sure to consider whether fixing the header file will break some other program that relies on the behaviour. In addition, you should report the bugs to your vendor and remember to re-apply the updates when you install a newer version of the operating system.
- Use the system header files, but add the missing definitions in local header files, or, worse, in the individual source files. This is a particularly obnoxious "solution",

especially when, as so often, the declarations are not dependent on a particular *ifdef*. In almost any system with reasonably complete header files there will be discrepancies between the declarations in the system header files and the declarations in the package. Even if they are only cosmetic, they will stop an ANSI compiler from compiling. For example, your system header files may declare `getpid` to return `pid_t`, but the package declares it to return `int`.

About the only legitimate use of this style of "fixing" is to declare functions that will really cause incorrect compilation if you don't declare them. Even then, declare them only inside an *ifdef* for a specific operating system. In the case of `getpid`, you're better off not declaring it: the compiler will assume the correct return values. Nevertheless, you will see this surprisingly often in packages that have already been ported to a number of operating systems, and it's one of the most common causes of porting problems.

- Make your own copies of the header files and use them instead. This is the worst idea of all: if anything changes in your system's header files, you will never find out about it. It also means you can't give your source tree to somebody else: in most systems, the header files are subject to copyright.