How to Use the Plan 9 C Compiler

Rob Pike

Introduction

The C compiler on Plan 9 is a wholly new program; in fact it was the first piece of software written for what would eventually become Plan 9 from Bell Labs. Programmers familiar with existing C compilers will find a number of differences in both the language the Plan 9 compiler accepts and in how the compiler is used.

The compiler is really a set of compilers, one for each architecture — MIPS, SPARC, Motorola 68020, Intel 386, etc. — that accept a dialect of ANSI C and efficiently produce fairly good code for the target machine. There is a packaging of the compiler that accepts strict ANSI C for a POSIX environment, but this document instead focuses on the native Plan 9 environment, that in which all the system source and almost all the utilities are written.

Source

The language accepted by the compilers is the core ANSI C language with some modest extensions, a greatly simplified preprocessor, a smaller library that includes system calls and related facilities, and a completely different structure for include files.

Official ANSI C accepts the old (K&R) style of declarations for functions; the Plan 9 compilers are more demanding. Without an explicit run-time flag (-B) whose use is discouraged, the compilers insist on new-style function declarations, that is, prototypes for function arguments. The function declarations in the libraries' include files are all in the new style so the interfaces are checked at compile time. For C programmers who have not yet switched to function prototypes the clumsy syntax may seem repellent but the payoff in stronger typing is substantial. Those who wish to import existing software to Plan 9 are urged to use the opportunity to update their code.

The compilers include an integrated preprocessor that accepts the familiar #include, #define for macros both with and without arguments, #undef, #line, #ifdef, #ifndef, and #endif. It supports neither #if nor ## and honors a single #pragma. The #if directive was omitted because it greatly complicates the preprocessor, is never necessary, and is usually abused. Conditional compilation in general makes code hard to understand; the Plan 9 source uses it very sparingly. Also, because the compilers remove dead code, regular if statements with constant conditions are more readable equivalents to many #ifs. To compile imported code ineluctably fouled by #if there is a separate command, /bin/cpp, that implements the complete ANSI C preprocessor specification.

Include files fall into two groups: machine-dependent and machine-independent. The machine-independent files occupy the directory /sys/include; the others are placed in a directory appropriate to the machine, such as /mips/include. The compiler searches for include files first in the machine-dependent directory and then in the machine-independent directory. At the time of writing there are eighteen machine-independent include files and three (per machine) machine-dependent ones: <ureg.h>, <stdarg.h>, and <u.h>. The first describes the layout of registers on the system stack, for use by the debugger; the second, as in ANSI C, defines a portable way to declare variadic functions. The third defines some architecture-dependent types such as jmp_buf for setjmp and also a set of typedef abbreviations for unsigned short and so on.

Here is an excerpt from /68020/include/u.h:

```
typedef unsigned short  ushort;
typedef unsigned char   uchar;
typedef unsigned long   ulong;
typedef unsigned int    uint;
typedef   signed char   schar;
typedef long            vlong;

typedef long    jmp_buf[2];
#define JMPBUFSP        0
#define JMPBUFPC        1
#define JMPBUFDPC       0
```

The type `vlong` is the largest integer type available; on some architectures it is a 64-bit value. The `#define` constants permit an architecture-independent (but compiler-dependent) implementation of stack-switching using `setjmp` and `longjmp`.

Every Plan 9 C program begins

```
#include <u.h>
```

because all the other installed header files use the `typedefs` declared in `<u.h>`.

In strict ANSI C, include files are grouped to collect related functions in a single file: one for string functions, one for memory functions, one for I/O, and none for system calls. Each include file is protected by an `#ifdef` to guarantee its contents are seen by the compiler only once. Plan 9 takes a completely different approach. Other than a few include files that define external formats such as archives, the files in `/sys/include` correspond to *libraries.* If a program is using a library, it includes the corresponding header. The default C library comprises string functions, memory functions, and so on, largely as in ANSI C, some formatted I/O routines, plus all the system calls and related functions. To use these functions, one must `#include` the file `<libc.h>`, which in turn must follow `<u.h>`, to define their prototypes for the compiler. Here is the complete source to the traditional first C program:

```
#include <u.h>
#include <libc.h>

void
main(void)
{
        print("hello world\n");
        exits(0);
}
```

The `print` routine and its relatives `fprint` and `sprint` resemble the similarly-named functions in Standard I/O but are not attached to a specific I/O library. In Plan 9 `main` is not integer-valued; it should call `exits`, which takes a string (or null; here ANSI C promotes the 0 to a `char*`) argument. All these functions are, of course, documented in the Programmer's Manual.

To use `printf`, `<stdio.h>` must be included to define the function prototype for `printf`:

```
#include <u.h>
#include <libc.h>
#include <stdio.h>

void
main(int argc, char *argv[])
{
        printf("%s: hello world with %d arguments\n", argv[0], argc-1);
        exits(0);
}
```

In fact Standard I/O is not used much in Plan 9. I/O libraries are discussed in a later section of this document.

There are libraries for handling regular expressions, bitmap graphics, windows, and so on, and each has an associated include file. The manual for each library states which include files are needed. The files are not protected against multiple inclusion and themselves contain no nested `#includes`. Instead the programmer is expected to sort out the requirements and to `#include` the necessary files once at the top of each source file. In practice this is trivial: this way of handling include files is so straightforward that it is rare for a source file to contain more than half a dozen `#includes`.

The compilers do their own register allocation so the `register` keyword is ignored. For different reasons, `volatile` and `const` are also ignored.

To make it easier to share code with other systems, Plan 9 has a version of the compiler, `pcc`, that provides the standard ANSI C preprocessor, headers, and libraries with POSIX extensions. `Pcc` is recommended only when broad external portability is mandated. It compiles slower, produces slower code (it takes extra work to simulate POSIX on Plan 9), eliminates those parts of the Plan 9 interface not related to POSIX, and illustrates the clumsiness of an environment designed by committee. `Pcc` is described in more detail in *''APE—The ANSI/POSIX Environment''*, by Howard Trickey.

Process

Each CPU architecture supported by Plan 9 is identified by a single, arbitrary, alphanumeric character: `v` for MIPS, `k` for SPARC, `z` for Hobbit, `2` for Motorola 68020 and 68040, `8` for Intel 386, and `6` for Intel 960. (This list is sure to grow.) The character labels the support tools and files for that architecture. For instance, for the 68020 the compiler is `2c`, the assembler is `2a`, the link editor/loader is `2l`, the object files are suffixed `.2`, and the default name for an executable file is `2.out`. Before we can use the compiler we therefore need to know which machine we are compiling for. The next section explains how this decision is made; for the moment assume we are building 68020 binaries and make the mental substitution for `2` appropriate to the machine you are actually using.

To convert source to an executable binary is a two-step process. First run the compiler, `2c`, on the source, say `file.c`, to generate an object file `file.2`. Then run the loader, `2l`, to generate an executable `2.out` that may be run (on a 680X0 machine):

```
2c file.c
2l file.2
2.out
```

The loader automatically links with whatever libraries the program needs, usually including the standard C library as defined by `<libc.h>`. Of course the compiler and loader have lots of options, both familiar and new; see the manual for details. The compiler does not generate an executable automatically; the output of the compiler must be given to the loader. Since most compilation is done under the control of `mk` (see below), this is rarely an inconvenience.

The distribution of work between the compiler and loader is unusual. The compiler integrates preprocessing, parsing, register allocation, code generation and some assembly. Combining these tasks in a single program is part of the reason for the compiler's efficiency. The loader does instruction selection, branch folding, instruction scheduling, and writes the final executable. There is no separate C preprocessor and no assembler in the usual pipeline. Instead the intermediate object file (here `.2` file) is a type of binary assembly language, similar to very regular assembly language but in binary form for quick I/O. The instructions in the intermediate format are not exactly those in the machine. For example, on the 68020 the object file may specify a MOVE instruction but the loader will decide just which variant of the MOVE instruction — MOVE immediate, MOVE quick, MOVE address, etc. — is most efficient.

The assembler, `2a`, is just a translator between the textual and binary representations of the object file format. It is not an assembler in the traditional sense. It has limited macro capabilities (the same as the integral C preprocessor in the compiler), clumsy syntax, and minimal error checking. For instance, the assembler will accept an instruction (such as memory-to-memory MOVE on the MIPS) that the machine does not actually support; only when the output of the assembler is passed to the loader will the error be discovered. The assembler is intended only for writing things, such as the machine-dependent part of an operating system, that need access to instructions invisible from C; very little code in Plan 9 is in assembly language.

The compilers take an option `-S` that causes them to print on their standard output the generated code in a format acceptable as input to the assemblers. This is of course merely a formatting of the data in the object file; therefore the assembler is just an ASCII-to-binary converter for this format. Near the end of this document is a brief introduction to the format of the assembly language. Other than the specific instructions, the input to the assemblers is largely architecture-independent.

The loader is an integral part of the compilation process. Each library header file contains a `#pragma` that tells the loader the name of the associated archive; it is not necessary to tell the loader which libraries a program uses. The C run-time startup is found, by default, in the C library. The loader starts with an undefined symbol, `_main`, that is resolved by pulling in the run-time startup code from the library. (The loader undefines `_mainp` when profiling is enabled, to force loading of the profiling start-up instead.)

Unlike its counterpart on other systems, the Plan 9 loader rearranges data to optimize access. This means the order of variables in the loaded program is unrelated to its order in the source. Most programs don't care, but some assume that, for example, the variables declared by

```
int a;
int b;
```

will appear at adjacent addresses in memory. On Plan 9, they won't.

Heterogeneity

When the system starts or a user logs in the environment is configured so the appropriate binaries are available in `/bin`. The configuration process is controlled by an environment variable, `$cputype`, with value such as `mips`, `68020`, or `sparc`. For each architecture there is a directory in root, with the appropriate name, that holds the binary and library files for that architecture. Thus `/mips/lib` contains the object code libraries for MIPS programs, `/mips/include` holds MIPS-specific include files, and `/mips/bin` has the MIPS binaries. These binaries are attached to `/bin` at boot time by binding `/$cputype/bin` to `/bin`, so `/bin` always contains the correct files.

The MIPS compiler, `vc`, by definition produces object files for the MIPS architecture, regardless of the architecture of the machine on which the compiler is running. There is a version of `vc` compiled for each architecture: `/mips/bin/vc`, `/68020/bin/vc`, `/sparc/bin/vc`, and so on, each capable of producing MIPS object files regardless of the native instruction set. If one is running on a SPARC, `/sparc/bin/vc` will compile programs for the MIPS; if one is running on machine `$cputype`, `/$cputype/bin/vc` will compile programs for the MIPS.

In Plan 9 the directory `/$cputype/bin` is bound to `/bin` so that the appropriate version of commands such as `date` are found by the shell: the shell looks for `date` in `/bin/date` and automatically finds the file `/$cputype/bin/date`. Therefore the MIPS compiler is known as just `vc`; the shell will invoke `/bin/vc` and that is guaranteed to be the version of the MIPS compiler appropriate for the machine running the command. Regardless of the architecture of the compiling machine, `/bin/vc` is *always* the MIPS compiler.

Also, the output of `vc` and `vl` is completely independent of `$cputype`: `.v` files compiled (with `vc`) on a SPARC may be linked (with `vl`) on a 386. (The resulting `v.out` will run, of course, only on a MIPS.) Similarly, the MIPS libraries in `/mips/lib` are suitable for loading with `vl` on any machine; there is only one set of MIPS libraries, not one set for each architecture that supports the MIPS compiler. (In other systems it is common for the intermediate format to be compiling-machine-dependent.)

Heterogeneity and `mk`

Most software on Plan 9 is compiled under the control of `mk`, a descendant of `make` that is documented in the Programmer's Manual. A convention used throughout the `mkfiles` makes it easy to compile the source into binary suitable for any architecture.

The variable `$cputype` is advisory: it reports the architecture of the current environment, and should not be modified. A second variable, `$objtype`, is used to set which architecture is being *compiled* for. The value of `$objtype` can be used by a `mkfile` to configure the compilation environment.

In each machine's root directory there is a short `mkfile` that just defines a set of macros for the compiler, loader, etc. Here is /mips/mkfile:

```
CC=vc
LD=vl
O=v
RL=rl
AS=va
OS=2kovz86
CPUS=mips 68020 sparc 386 hobbit
CFLAGS=
```

CC is obviously the compiler, AS the assembler, and LD the loader. O is the suffix for the object files and RL is the program that inserts a table of contents into a library file (a dreg, really; the program rl works for all supported architectures). CPUS and OS are used in special rules described below.

Here is a mkfile to build the installed source for sam:

```
</$objtype/mkfile
OBJ=sam.$O address.$O buffer.$O cmd.$O disc.$O error.$O file.$O \
        io.$O list.$O mesg.$O moveto.$O multi.$O plan9.$O rasp.$O regexp.$O \
        string.$O sys.$O xec.$O

$O.out: $OBJ
        $LD $OBJ

install:        $O.out
        cp $O.out /$objtype/bin/sam

installall:
        for(objtype in $CPUS) mk install

%.$O:   %.c
        $CC $CFLAGS $stem.c

$OBJ:                                           sam.h errors.h mesg.h
address.$O cmd.$O parse.$O xec.$O unix.$O:      parse.h

clean:V:
        rm -f [$OS].out *.[$OS] y.tab.?
```

(The actual mkfile imports most of its rules from other secondary files, but this example works and is not misleading.) The first line causes mk to include the contents of /$objtype/mkfile in the current mkfile. If $objtype is mips, this inserts the macro definitions above into the mkfile. In this case the rule for $O.out will therefore use the MIPS tools to build v.out. The %.$O rule in the file uses mk's pattern matching facilities to connect the source files to the object files through a run of the compiler. (The text of the rules is passed directly to the shell, rc, without further translation. See the mk manual if any of this is unfamiliar.) Because the default rule builds $O.out rather than sam, it is possible to maintain binaries for multiple machines in the same source directory without conflict. This is also, of course, why the output files from the various compilers and loaders has distinct names.

The rest of the mkfile should be easy to follow; notice how the rules for clean and installall (that is, install versions for all architectures) use other macros defined in /$objtype/mkfile. In Plan 9, mkfiles for commands conventionally contain rules to install (compile and install the version for $objtype), installall (compile and install for all $objtypes), and clean (remove all object files, binaries, etc.).

The mkfile is easy to use. To build a MIPS binary, v.out:

```
% objtype=mips
% mk
```

To build and install a MIPS binary :

```
% objtype=mips
% mk install
```

To build and install all versions:

```
% mk installall
```

These conventions make cross-compilation as easy to manage as traditional native compilation. Plan 9 programs compile and run without change on machines from large mulitprocessors to laptops.

Portability

Within Plan 9, it is painless to write portable programs, programs whose source is independent of the machine on which they execute. The operating system is fixed and the compiler, headers and libraries are constant so most of the stumbling blocks to portability are removed. Attention to a few details can avoid those that remain.

Plan 9 is a heterogeneous environment, so programs must *expect* that external files will be written by programs on machines of different architectures. The compilers, for instance, must handle without confusion object files written by other machines. The traditional approach to this problem is to pepper the source with `#ifdefs` to turn byte-swapping on and off. Plan 9 takes a different approach; there are no machine-dependent `#ifdefs` in any of the source. Instead the programs read and write files in a defined format, either (for low volume applications) as formatted text, or (for high volume applications) as binary in a known byte order. If the external data was written with the most significant byte first, the following code will read a 4-byte integer correctly regardless of the architecture of the executing machine (assuming an unsigned long will hold 4 bytes):

```
ulong
getlong(void)
{
	ulong l;

	l = (getchar()&0xFF)<<24;
	l |= (getchar()&0xFF)<<16;
	l |= (getchar()&0xFF)<<8;
	l |= (getchar()&0xFF)<<0;
	return l;
}
```

Note that this code does not ‘swap’ the bytes; instead it just reads them in the correct order. Variations of this code will handle any binary format and also avoid problems involving how structures are padded, how words are aligned, and other impediments to portability. Be aware, though, that extra care is needed to handle floating point data.

Efficiency hounds will argue that this method is unnecessarily slow and clumsy when the executing machine has the same byte order (and padding and alignment) as the data. I/O speed is rarely the bottleneck for an application, however, and the gain in simplicity of porting and maintaining the code greatly outweighs the minor speed loss from handling data in this general way. This method is how the Plan 9 compilers, the window system, and even the file servers transmit data between programs.

To port programs beyond Plan 9, where the system interface is more variable, it is probably necessary to use `pcc` and hope that the target machine supports ANSI C and POSIX.

I/O

The default C library, defined by the include file `<libc.h>`, contains no buffered I/O package. It does have several entry points for printing formatted text: `print` outputs text to the standard output, `fprint` outputs text to a specified integer file descriptor, and `sprint` places text in a character array. But to access library routines for buffered I/O, a program must explicitly mention an appropriate library.

The recommended I/O library, used by most Plan 9 utilities, is `bio` (buffered I/O), defined by `<bio.h>` and accessed by appending `-lbio` to the command line for the loader. There also exists an implementation of ANSI Standard I/O, `stdio`.

`Bio` is small and very efficient, particularly for buffer-at-a-time or line-at-a-time I/O. Even for character-at-a-time I/O, however, it is significantly faster than the Standard I/O library, `stdio`. Its interface is compact and regular, although it lacks a couple of conveniences. The most noticeable is that one must explicitly define buffers for standard input and output; `bio` does not predefine them. Here is a program to copy input to output a character at a time using `bio`:

```
#include <u.h>
#include <libc.h>
#include <bio.h>

Biobuf  bin;
Biobuf  bout;

main(void)
{
        int c;

        Binit(&bin, 0, OREAD);
        Binit(&bout, 1, OWRITE);

        while((c=Bgetc(&bin)) != Beof)
                Bputc(&bout, c);
        exits(0);
}
```

For peak performance, we could replace `Bgetc` and `Bputc` by their equivalent in-line macros `BGETC` and `BPUTC` but the performance gain would be modest. For more information on `bio`, see the Programmer's Manual.

Perhaps the most dramatic difference in the I/O interface of Plan 9 from other systems' is that text is not ASCII. The format for text in Plan 9 is a byte-stream encoding of 16-bit characters. The character set is based on Unicode and is backward compatible with ASCII: characters with value 0 through 127 are the same in both sets. The 16-bit characters, called *runes* in Plan 9, are encoded using a representation called UTF, a encoding we proposed and expect to be accepted by X-Open. UTF defines multibyte sequences to represent character values from 0 to 65535. In UTF, character values up to 127 decimal, 7F hexadecimal, represent themselves, so straight ASCII files are also valid UTF. Also, UTF guarantees that bytes with values 0 to 127 (NUL to DEL, inclusive) will appear only when they represent themselves, so programs that read bytes looking for plain ASCII characters will continue to work. Any program that expects a one-to-one correspondence between bytes and characters will, however, need to be modified. An example is parsing file names. File names, like all text, are in UTF, so it is incorrect to search for a character in a string by `strchr(filename, c)` because the character might have a multi-byte encoding. The correct method is to call `utfrune(filename, c)`, defined in *rune*(2), which interprets the file name as a sequence of encoded characters rather than bytes. In fact, even when you know the character is a single byte that can represent only itself, it is safer to use `utfrune` because that assumes nothing about the character set and its representation. The representation may change—Plan 9 has already changed it several times.

The library defines several symbols relevant to the representation of characters. Any byte with unsigned value less than `Runesync` will not appear in any multi-byte encoding of a character. `Utfrune` compares the character being searched against `Runesync` to see if it is sufficient to call `strchr` or if the byte stream must be interpreted. Any byte with unsigned value less than `Runeself` is represnted by a single byte with the same value. Finally, when errors are encoutered converting to runes from a byte stream, the library returns the rune value `Runeerror` and advances a single byte. This permits programs to find runes embedded in binary data.

`Bio` includes routines `Bgetrune` and `Bputrune` to transform the external byte stream UTF format to and from internal 16-bit runes. Also, the `%s` format to `print` accepts UTF; `%c` prints a character after narrowing it to 8 bits. The `%S` format prints a null-terminated sequence of runes; `%C` prints a character after narrowing it to 16 bits. For more information, see the Programmer's Manual, in particular *utf*(6) and *rune*(2); there is not room for the full story here.

These issues affect the compiler in several ways. First, the C source is in UTF. Although C variables are formed from ASCII alphanumerics, comments and literal strings may contain any characters encoded in UTF. The declaration

```
char *cp = "abcÿ";
```

initializes the variable `cp` to point to an array of bytes holding the UTF representation of the characters `abcÿ.` The type `Rune` is defined in `<u.h>` to be `ushort`, which is also the 'wide character' type in the compiler. Therefore the declaration

```
Rune *rp = L"abcÿ";
```

initializes the variable `rp` to point to an array of unsigned short integers holding the 16-bit values of the characters `abcÿ`. Note that in both these declarations the characters in the source that represent `abcÿ` are the same; what changes is how those characters are represented in memory in the program. The following two lines:

```
print("%s\n", "abcÿ");
print("%S\n", L"abcÿ");
```

produce the same UTF string on their output, the first by copying the bytes, the second by converting from runes to bytes.

In C, character constants are integers but narrowed through the `char` type. The Unicode character ÿ has value 255, so if the `char` type is signed, the constant `'ÿ'` has value -1 (which is equal to EOF). On the other hand, `L'ÿ'` narrows through the wide character type, `ushort`, and therefore has value 255.

Arguments

Defined in `<libc.h>` are some macros for parsing the arguments to `main()`. They are described in *ARG*(2) but are fairly self-explanatory. There are four macros: `ARGBEGIN` and `ARGEND` are used to bracket a hidden `switch` statement within which `ARGC` returns the current option character (rune) being processed and `ARGF` returns the argument to the option, as in the loader option `-o file`. Here, for example, is the beginning of `main()` in `mount.c` (see *bind*(1)) that cracks its arguments:

```
void
main(int argc, char *argv[])
{
        char *spec, *srv;
        ulong flag = 0;

        srv = "";
        ARGBEGIN{
        case 'a':
                flag |= MAFTER;
                break;
        case 'b':
                flag |= MBEFORE;
                break;
        case 'c':
                flag |= MCREATE;
                break;
        case 't':
                srv = "any";
                break;
        case 's':
                srv = ARGF();
                if(srv == 0)
                        usage();
                break;
        default:
                usage();
        }ARGEND
        if(argc == 2)
                spec = "";
        else if(argc == 3)
                spec = argv[2];
        else
                usage();
```

Extensions

The compiler has several extensions to ANSI C, all of which are used extensively in the system source. First, *structure displays* permit `struct` expressions to be formed dynamically. Given these declarations:

```
typedef struct Point Point;
typedef struct Rectangle Rectangle;

struct Point
{
        int x, y;
};

struct Rectangle
{
        Point min, max;
};

Point   p, q, add(Point, Point);
Rectangle r;
int     x, y;
```

this assignment may appear anywhere an assignment is legal:

```
r = (Rectangle){add(p, q), (Point){x, y+3}};
```

The syntax is the same as for initializing a structure but with a leading cast.

If an *anonymous structure* or *union* is declared within another structure or union, the members of the internal structure or union are addressable without prefix in the outer structure. This feature eliminates the clumsy naming of nested structures and, particularly, unions. For example, after these declarations,

```
struct Lock
{
        int     locked;
};

struct Node
{
        int     type;
        union{
                double  dval;
                double  fval;
                long    lval;
        };
        struct Lock;    /* anonymous structure */
} *node;

void    lock(struct Lock*);
```

one may refer to `node->type`, `node->dval`, `node->fval`, `node->lval`, and `node->locked`. Moreover, the address of a `struct Node` may be used without a cast anywhere that the address of a `struct Lock` is used, such as in argument lists. The compiler will automatically promote the type and adjust the address. Thus one may invoke `lock(node)`.

Anonymous structures and unions may be accessed by type name if (and only if) they are declared using a `typedef` name. For example, using the above declaration for `Point`, one may declare

```
struct
{
        int     type;
        Point;
}p;
```

and refer to `p.Point`.

In the initialization of arrays, a number in square brackets before an element sets the index for the initialization. For example, to initialize some elements in a table of function pointers indexed by ASCII character,

```
void    percent(void), slash(void);

void    (*func[128])(void) =
{
        ['%']   percent,
        ['/']   slash,
};
```

Finally, the declaration

```
extern register reg;
```

(*this* appearance of the register keyword is not ignored) allocates a global register to hold the variable `reg`. External registers must be used carefully: they need to be declared in *all* source files and libraries in the program to guarantee the register is not allocated temporarily for other purposes. Especially on machines with few registers, such as the i386, it is easy to link accidentally with code that has already usurped the global registers and there is no diagnostic when this happens. Used wisely, though, external registers are powerful. The Plan 9 operating system uses them to access per-process and per-machine data structures on a multiprocessor. The storage class they provide is hard to create in other ways.

The compile-time environment

The code generated by the compilers is 'optimized' by default: variables are placed in registers and peephole optimizations are performed. The compiler flag `-N` disables these optimizations. Registerization is done locally rather than throughout a function: whether a variable occupies a register or the memory location identified in the symbol table depends on the activity of the variable and may change throughout the life of the variable. The `-N` flag is rarely needed; its main use is to simplify debugging. There is no information in the symbol table to identify the registerization of a variable, so `-N` guarantees the variable is always where the symbol table says it is.

Another flag, `-w`, turns *on* warnings about portability and problems detected in flow analysis. Most code in Plan 9 is compiled with warnings enabled; these warnings plus the type checking offered by function prototypes provide most of the support of the Unix tool `lint` more accurately and with less chatter. Two of the warnings, 'used and not set' and 'set and not used', are almost always accurate but may be triggered spuriously by code with invisible control flow, such as in routines that call `longjmp`. The compiler statements

```
SET(v1);
USED(v2);
```

decorate the flow graph to silence the compiler. Either statement accepts a comma-separated list of variables. Use them carefully; they may silence real errors.

Debugging

The debugger on Plan 9 is `db`, a revision of Unix `adb` that supports multiple architectures in a single program. There is room for a good source-level debugger but until the compiler emits a symbol table with types `db` will have to serve.

`Db` has several improvements over `adb`. Most important is that it reports C source line numbers with the `z` command. Breakpointing, and in particular conditional breakpointing, work well. Finally, it supports cross-architecture debugging comfortably.

For simple debugging, however, `db` is fine. Imagine a program has crashed mysteriously:

```
% X11/X
Fatal server bug!
failed to create default stipple
X 106: suicide: sys: trap: fault read addr=0x0 pc=0x00060a82
%
```

When a process dies on Plan 9 it hangs in the 'broken' state for debugging. `Db` takes the process id of the broken process and reports the type of the binary, the last trap it executed, and the failing instruction:

```
% db 106
68020 binary
bus error
abort.c:8 abort+#2?                     TSTL    #0($0)
```

The `$C` (capital `C`) command to `db` reports the stack traceback:

```
$C
abort() abort.c:4 called from FatalError+#4e misc.c:421
FatalError(s9=#e02, s8=#4901d200, s7=#2, s6=#72701, s5=#1,
    s4=#7270d, s3=#6, s2=#12, s1=#ff37f1c, s0=#6, f=#7270f)
    misc.c:416 called from gnotscreeninit+#4ce gnot.c:792
        gnotscreeninit.buf/   #4901b300
gnotscreeninit(snum=#0, sc=#80db0) gnot.c:766 called from
    AddScreen+#16e main.c:610
AddScreen(argv=#ff37fec, argc=#1, pfnInit=#23bc) main.c:531
    called from InitOutput+#16 gnot.c:924
        AddScreen.i/          #0
        AddScreen.pScreen/    #80db0
InitOutput(argv=#ff37fec, argc=#1, si=#72cd4) gnot.c:920
    called from main+#1e2 main.c:227
main(argc=#1, argv=#ff37fec) main.c:137 called from _main+#1e
    plan968020/MAIN.s:8
        main.alwaysCheckForInput/       #0
        main.i/     #80
_main(inargv=#ff37ff6, inargc=#1) plan968020/MAIN.s:1 called from #1
```

The command `$c` (lower case `c`) omits the local variables.

The compiler does not place in the executable information describing the types of variables, but a compile-time flag provides crude support for symbolic debugging. The `-s` flag to the compiler suppresses code generation and identifies a structure or union in the program whose layout is to be printed on standard output. The command `dbfmt` translates this output into `db` commands to display the data structure. The easiest way to use this feature is to put a rule in the mkfile:

```
%.db:   main.$O
        $CC -s$stem main.c | dbfmt > $stem.db
```

Making `Dir.db` for example will produce a file by that name containing commands to print the contents of a `struct Dir` (a type in `<libc.h>`). The `$<` command of `db` will read the file as commands. For example,

```
main.dirbuf$<Dir.db
```

will format the contents of `main`'s local variable `dirbuf`. These format files can of course be edited to improve the output.

For more information, see the `db` manual page.