CS 536 — Spring 1998

Make

One of the problems in producing and maintaining a large program is the amount of time required to compile it. Splitting a program into separately compiled modules alleviates this problem, but also introduces a new problem: keeping track of which modules need to be recompiled. This task appears to be tedious but simple — recompile those modules that have been changed. However, modules often have a complex relationship. For example, if a module is changed, all programs and modules that use it must be recompiled.

Having determined which modules need updating, we have yet another problem (though it is simpler): remembering *how* to update them. The program may not consist merely of C modules, but also may use tables from a parser generator, some assembly language routines for low-level interfaces, and routines from a non-standard library. Keeping track of all the different options and calling conventions is a tedious and error-prone chore.

Make is a tool available under Unix that takes care of this updating process. In normal usage, a special *makefile* nestles among the source files of the program, containing a description of the dependencies among files, and the procedures used to update them. Make uses this dependency information and the last modified time kept by Unix to determine which files are out of date, then performs the updating procedures specified for each.

The objects that the make program deals with are files; they may contain source code, object code, parsing tables or even a mailing list. All make knows about a file is the dependency and updating rules contained in the makefile. Thus, make is very language independent, updating C programs, Pascal programs and bibliographies with equal ease. On the other hand, it is incapable of handling subtle points of dependency. There is no direct way to specify, for example, that a change to a comment in the body of a module can in no way affect any other program or module. As far as make is concerned, changing a comment is the same as changing the type of an

```
prog: main.o step1.o step2.o dataaccess.o
        gcc -o prog main.o step1.o step2.o dataaccess.o

main.o: main.c step1.h step2.h dataaccess.h
        gcc -c main.c

step1.o: step1.c step1.h dataaccess.h
        gcc -c step1.c

step2.o: step2.c step2.h  dataaccess.h
        gcc -c step2.c

dataaccess.o: dataaccess.c dataaccess.h
        gcc -c dataaccess.c
```

Figure 1 — A sample makefile

important variable. The dependency rules and the time of last modification are the two important criteria.

To use make, we must program a makefile defining the dependencies among program modules, and how source files are to be translated into object files. The language used to program makefiles isn't particularly difficult; a sample makefile is shown in Figure 1.

In general, a makefile contains a rule specifying the files an object depends on, and a command that can be used to build the desired object. We start with a rule for the executable program we wish to build. Call this prog. Prog depends on the four object files main.o, step1.o, step2.o, and dataaccess.o. Prog is built by a suitable call to gcc, with these files as arguments. The exact format is

```
prog: main.o step1.o step2.o dataaccess.o
	gcc –o prog main.o step1.o step2.o dataaccess.o
```

The first line defines the object we wish to build, followed by a ":" and a list of the files it depends upon. The command to build the object follows on the next line, indented with a tab (the tab *must* be used to precede the desired command). We next need rules on how to build each of the ".o" files. For example, step1.o depends on step1.c (the source file) *and* dataaccess.h (because it includes this header file). Step1.o is built by simply compiling its source file. Hence the rule is

```
step1.o: step1.c dataaccess.h
	gcc –c step1.c
```

Similar rules are needed for main.o, step2.o, and dataaccess.o. Once we have created a makefile (in the current directory), we are ready to use make. Assume that no object files (suffixed with ".o") exist and that no executable exists. We just type the command "make." Make responds by showing the steps needed to build prog, and then performs them:

```
gcc –c dataaccess.c
dataaccess.c:
gcc –c step1.c
step1.c:
gcc –c step2.c
step2.c:
gcc –c main.c
main.c:
gcc –o prog main.o step1.o step2.o dataaccess.o
```

That is, each of the source files (suffixed with .c) is compiled, then the object files are linked to produce the executable, prog. Note also that the files are automatically compiled in the correct order—first dataaccess.c, then step1.c and step2.c, and finally main.c.

If we invoke make again, it responds:

```
'prog' is up to date.
```

That is, nothing needs to be recompiled!

Now assume we edit step1.c. We then invoke make, and it responds:

```
gcc –c step1.c
step1.c:
gcc –o prog main.o step1.o step2.o dataaccess.o
```

Since step1.o depends on step1.c, it is recompiled, then the object modules are relinked. Note that since neither main.o nor step2.c depends on step1.c in any way, they are *not* recompiled. Make is smart enough to track what has been changed and what depends on what, and it only performs the steps necessary to create an up-to-date executable program.

In general an updated makefile is needed only when new modules are added or new uses of modules are created (i.e., new includes are added).

More on Makefiles

A rule in a makefile has the general form

```
target-file : generator1 ... generatorN
        commands
```

which specifies that target-file depends on the files generator1 ,..., generatorN, and that performing commands will correctly update target-file. Thus in our example,

```
main.o: main.c step1.h step2.h dataaccess.h
        gcc –c main.c
```

Main.o depends on four files, and it can be built by asking gcc to compile file main.c.

Generators may appear as the target of other rules; thus, the rules describe a graph of dependencies. Make searches this graph, looking for out of date files. (Non-existent files are assumed to be out of date.)

Make provides for the use of implicit updating rules; this simplifies the description of the normal situation, and allows one to concentrate on the exceptions. Implicit rules assume that files follow the Unix ".suffix" convention (".c" for C programs, ".p" for Pascal programs, ".o" for object files, etc.). An implicit rule has the form

```
.x.y :
        commands
```

which specifies commands to be performed to make a ".y" file out of the corresponding ".x" file. In Pascal, the rules

```
.p.o:
        pc –c $*.p
```

state that to create an object file (a ".o" file) from a Pascal source file unit (a ".p" file), the Pascal compiler should process the compilation unit with the appropriate options.

Make also knows standard implicit rules for common Unix applications such as C programs and YACC grammars. Make is a fairly general, reasonably simple tool for incremental updating. It does not address certain problems, such as maintaining multiple versions, nor does it handle specialized dependencies possible in languages such as Ada. It is, understandably, somewhat biased towards the needs of a C programmer on a Unix system.

Common C/C++ Errors

(1) = is the assignment operator; == is the equal comparison operator. The statement

```
if (a=b)
```

is often legal but almost always wrong. It assigns b to a and tests if the assigned value is 0 (ie, false) or non-0 (i.e. true).

(2) Multi-dimensional arrays are indexed as a[i][j]; **not** a[i,j]. a[a,j] is legal but returns a pointer to a subarray.

(3)
```
for (a=0; a<b; a++);
  {
   ...
  }
```
will iterate over the **null** statement (because of the ";" on the first line)

(4) Functions of no arguments are called as f(). Writing just f will evaluate a pointer to f and then throw it away!

(5) Often the number and/or types of arguments to a call is unchecked, especially for externally defined subprograms with incomplete definitions.

(6) A `break` statement should follow each case alternative in a `switch` statement.

(7) Be careful of unexpected binding orders:

```
*p1->a   doesn't mean   *(p1->a)
```

Similarly, ++p1->a doesn't mean ++(p1->a)

(8) Given
```
  {
   float f;
   int i, j;

   ...
   f = i / j;
  }
```
f will always contain an integral value (no fraction). To get floating point division you should use

```
f = (float)i / j;
```

(9) Some functions (like scanf) require a pointer to a variable rather than the value of the variable. Thus

```
scanf("%d",var);
```

will likely core-dump, since var's value will be used as an address. You must use the "&" operator to extract var's address.

```
scanf("%d",&var);
```

is what should be used.

(10) In C arrays are 0 based. Thus given

```
int  ar[10];
```

valid subscripts are 0 to 9. 10 is never valid.

(11) Local variables (which have storage class automatic) and heap space obtained via malloc are uninitialized. (calloc can be used instead of malloc to obtain memory cleared to 0's).

(12) Arrays cannot be copied using the = operator (though structs can). Normally the name of an array is the same as a pointer to its 0-th element. Given

```
int  a[10], b[10];
```

The statement a=b won't copy b's data into a; it is an illegal attempt to assign b's starting address (&b[0]) to a's starting address. Since a's address is a constant this can't be done. Use

```
memcpy(a,b,sizeof(a));
```

instead.

(13) The and and or operators are && and ||. & and | are bitwise and and or. Thus

```
if (a | b)
```

or's together the bits of a and b and almost certainly gets a non-zero value interpreted as true.

(14) Declaring a pointer does not automatically allocate storage for the pointer. Initially, the pointer will contain `null` (often represented by 0) or garbage. To give a pointer a useful value (and make *p or p->f valid) you must assign the pointer a valid pointer value or call malloc to create new space.

(15) If a pointer is used in a subroutine to simulate a var parameter, be sure to use "*" or -> with the subroutine to access the data pointed to. Thus

```
void increment(int *val) {
      val++;
}
```

is wrong; (*val)++ should have been used. C++ references (e.g., int&) should be used to implement parameters that can be changed during a call.

(16) C gives you access to a vast and highly useful subroutine library. You should examine what's there and use library routines to ease your programming chores. Sometimes, though, library routines work in unexpected ways. C-style strings are typically defined as char *'s. Thus you can't compare strings via ==, since str1 == str2 checks whether the 2 strings are at the same memory address. Instead the strcmp routine should be used. But even this has pitfalls. You'd expect

```
if (strcmp("abc","abc"))
```

or

```
if (strcmp("abc","abc") == 1 )
```

to work, but they don't. strcmp does a 3 way compare, with a negative val denoting the first string is less, 0 denoting that are equal, and a positive value denoting the first string is greater. So strcmp("abc","abc") returns 0, even though 0 usually denotes false!

The String class in C++ avoids these problems and should be used instead of C-style char*'s.

(17) Characters and strings are different, even when the string contains a single character. 'c' is a character literal, an integer value denoting the character's ASCII code. "c" is a string literal, a pointer to a zero-delimited sequence of characters. The call strcmp(answer,'c') will probably bomb, passing the character code of c to strcmp, which will try to use it as a string address!

(18) |= and != look awfully much alike, especially on monitors with dead pixels! They sure don't mean the same thing though (|= or's bitwise the left and right operands and assigns the result to the left operand.

(19) A leading 0 in a literal means an octal rather than decimal literal. Thus 011 == 11 is false!