make Programming Discipline

make is a utility that relieves you of the drudgery of building a complex C program.

How does make work?

To understand how **make** works, it is first necessary to understand how a C program is built: how **Let's C** takes you from the C source code that you write to the executable program that you can run on your computer.

The file of C source code that you write is called a *source module*. When **Let's C** compiles a source module, it uses the C code in the source module, plus the code in the header files that the code calls to produce an *object module*. This object module is *not* executable by itself. To create an *executable file*, the object module generated from your source module must be handed to a linker, which links the code in the object module with the appropriate library routines that the object module calls, and adds the appropriate C runtime startup routine.

For example, consider the following C program, called **hello.c**:

```
main()
{
        printf("Hello, world\n");
}
```

When **Let's C** compiles the file that contains C code shown above, it generates an object module called **hello.obj**. This object module is not executable because it does not contain the code to execute the function **printf**; that code is contained in a library. To create an executable program, you must hand **hello.obj** to the linker **ld**, which copies the code for **printf** from a library and into your program, adds the appropriate C runtime startup routine, and writes the executable file called **hello.exe**. This third file, **hello.exe**, is what you can execute on your computer.

The term *dependency* describes the relationship of executable file to object module to source module. The executable program *depends* on the object module, the library, and the C runtime startup. The object module, in turn, depends on the source module and its header files (if any).

A program like **hello.exe** has a simple set of dependencies: the executable file is built from one object module, which in turn is compiled from one source module. If you changed the source module **hello.c**, creating an updated version of **hello.exe** would be easy: you would simply compile **hello.c** to create **hello.obj**, which you would link with the library and the runtime startup to create **hello.exe**. **Let's C**, in fact, does this for you automatically: all you need to do is type

```
cc hello.c
```

and **Let's C** takes care of everything.

On the other hand, the dependencies of a large program can be very complex. For example, the executable file for the MicroEMACS screen editor is built from several dozen object modules, each of which is compiled from a source module plus one or more header files. Updating a program as large as MicroEMACS, even when you change only one source module, can be quite difficult. To rebuild its executable file by hand, you must remember the names of all of the source modules used, compile them, and link them into the executable file. Needless to say, it is very inefficient to recompile several dozen object modules to create an executable when you have changed only one of them.

make automatically rebuilds large programs for you. You prepare a file, called a **makefile**, that describes your program's chain of dependencies. **make** then reads your **makefile**, checks to see which source modules have been updated, recompiles only the ones that have been changed, and then relinks all of the object modules to create a new executable file. **make** both saves you time, because it recompiles only the source modules that have changed, and spares you the drudgery of rebuilding your large program by hand.

Try make

The following example shows how easy it is to use **make**.

To begin, **make** examines the time and date that MS-DOS has stamped on each source file and object module. When you edit a source module, MS-DOS marks it with the time at which you edited it. Thus, if a source module has a time that is *later* than that of its corresponding object module, then **make** knows that the source module was changed since the object module was last compiled and it will compile a new object module from the altered source module. If you do not reset the time on your system whenever you reboot, *every time,* some files will not have the correct date and time and **make** cannot work correctly.

To see how **make** works, try compiling a program called **factor**. It is built from the following files:

```
atod.c
factor.c
makefile
```

All three are included with your copy of **Let's C**.

If you do not have a hard disk, insert disk 8 (which holds the sample programs) into drive B, and make sure that disk 2 (the compiler disk) is in drive A. Use the **cd** command to shift into directory **src**.

Now, type **make**. **make** will begin by reading **makefile**, which describes all of **factor**'s dependencies. It will then use the **makefile** description to create **factor**. The following will appear on your screen:

```
cc -c factor.c
cc -c atod.c
cc -f -o factor.exe factor.obj atod.obj -lm
```

Each of these messages describes an action that **make** has performed. The first shows that **make** is compiling **factor.c**, the second shows that it is compiling **atod.c**, and the third shows that it is linking the compiled object modules **atod.obj** and **factor.obj** to create the executable file **factor.exe**.

When **make** has finished, the MS-DOS prompt will return. To see how your newly compiled program works, type

```
factor 100
```

factor will calculate the prime factors of its argument **100**, and print them on the screen.

To see what happens if you try to re-make your file, type **make** again. **make** will run quietly for a moment, and then exit. **make** checked the dates and times of the object modules and their corresponding source modules and saw that the object modules had a time later than that of the source modules. Because no source module changed, there was no need to recompile an object module or relink the executable file, so **make** quietly exited.

To see what happens when one of the source modules changes, try the following. Use the MicroEMACS screen editor to open the file **factor.c** for editing. Insert the following line into the comments at the top, immediately following the **/***:

```
* This comment is for test purposes only.
```

Now exit. Type **make** once again. This time, you will see the following on your screen:

```
cc -c factor.c
cc -f -o factor.exe factor.obj atod.obj -lm
```

Because you altered the source module **factor.c**, its time was later than that of its corresponding object module, **factor.obj**. When **make** compared the times of **factor.c** and **factor.obj**, it noted that **factor.c** had been altered. It then recompiled **factor.c** and relinked **factor.obj** and **atod.obj** to re-create the executable file **factor.exe**. **make** did not touch the source module **atod.c** because **atod.c** had not been changed since the last time it was compiled.

As you can see, **make** greatly simplifies the construction of a C program that uses more than one source module.

Essential make

Although **make** is a powerful program, its basic features are easy to master. This section will show you how to construct elementary **make** scripts.

The makefile

When you invoke **make**, it searches the directories named in the environmental variable **PATH** for a file called **makefile**. As noted earlier, the **makefile** is a text file that describes a C program's dependencies. It also describes the type of program you wish to build, and the commands for building it.

A **makefile** has three basic parts.

First, the **makefile** describes the executable file's dependencies. That is, it lists the object modules needed to create the executable file. The name of the executable file is always followed by a colon ':' and then by the names of files from which the target file is generated.

For example, if the program **feud.exe** is built from the object modules **hatfield.obj** and **mccoy.obj**, you would type:

```
feud.exe:    hatfield.obj mccoy.obj
```

If the files **hatfield.obj** and **mccoy.obj** do not exist, **make** knows to create them from the source modules **hatfield.c** and **mccoy.c**.

Second, the **makefile** holds one or more *command* lines. The command line gives the command to compile the program in question. The only difference between a **makefile** command line and an ordinary **cc** command is that a **makefile** command line *must* begin with a space or a tab character.

For example, the **makefile** to generate the program **feud.exe** must contain the following command line:

```
        cc -o feud.exe hatfield.obj mccoy.obj
```

For a detailed description of the **cc** command and its options, refer to the entry for **cc** in the Lexicon.

Third, the **makefile** lists all of the header files that your program uses. These are given so that **make** can check if they were modified since your program was last compiled. For example, if the program **hatfield.c** used the header file **shotgun.h** and **mccoy.c** used the header files **rifle.h** and **pistol.h**, the **makefile** to generate **feud.exe** would include the following lines:

```
hatfield.obj: shotgun.h
mccoy.obj: rifle.h pistol.h
```

Thus, the entire **makefile** to generate the program **feud.exe** is as follows:

```
feud.exe: hatfield.obj mccoy.obj
        cc -o feud.exe hatfield.obj mccoy.obj

hatfield.obj: shotgun.h
mccoy.obj: rifle.h pistol.h
```

A **makefile** may also contain *macro definitions* and *comments*. These are described below.

Building a simple makefile

The program **factor.exe** is built from two source modules, **factor.c** and **atod.c**. No header files are used. The **makefile** contains the following two lines:

```
factor.exe: factor.obj atod.obj
        cc -f -o factor.exe factor.obj atod.obj -lm
```

The first line describes the dependency for the executable file **factor.exe** by naming the two object modules needed to build it. The second line gives the command needed to build **factor.exe**. The option **-lm** at the end of the command line tells **cc** that this program needs the mathematics library **libm** when the program is linked. No header file dependencies are described because these programs use no header files.

Comments and macros

You can embed comments within a **makefile**. A *comment* is a line of text that is ignored; this lets you "document" the file, so that whoever reads it will now know what it is for. **make** ignores all lines that begin with a pound sign '#'. For example, you may wish to include the following information in your **makefile** for **factor**:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.

        factor: factor.obj atod.obj
                cc -f -o factor.exe factor.obj atod.obj -lm
```

Anyone who reads this file will know immediately what it is for by looking at the comments.

make also lets you define macros within your **makefile**. A *macro* is a symbol that represents a string of text. Usually, a macro is defined at the beginning of the **makefile** using a *macro definition statement*. This statement uses the following syntax:

```
SYMBOL = string of text
```

Thereafter, when you use the symbol in your **makefile**, it must begin with a dollar sign '$' and be enclosed within parentheses.

Macros eliminate the chore of retyping long strings of file names. For example, with the **makefile** for the program **factor**, you may wish to use a macro to substitute for the names of the object modules out of which it is built. This is done as follows:

```
# This makefile generates the program "factor".
# "factor" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# "libm", but it requires no special header files.
# "-f" lets you use printf for floating-point numbers.
```

```
OBJ = factor.obj atod.obj
factor: $(OBJ)
        cc -o factor.exe $(OBJ) -lm
```

The macro **OBJ** is used in this **makefile**. If you use a macro that has not been defined, **make** substitutes an empty string for it. The use of a macro makes sense when generating large files out of a dozen or more source modules. You avoid retyping the source module names, and potential errors are avoided.

Setting the time

As noted above, **make** checks to see which source modules have been modified before it regenerates your C program. This is done to avoid wasteful recompiling of source modules that have not been updated.

make determines that a source module has been altered by comparing its date against that of the target program. For example, if the object module **factor.obj** was generated on March 16, 1987, 10:52:47 A.M., and the source module **factor.c** was modified on March 20, 1987, at 11:19:06 A.M., **make** will know that **factor.c** needs to be recompiled because it is *younger* than **factor.obj**.

For this reason, if you wish to use **make**, you *must* reset the date and time every time you reboot your system. Some users do not do this routinely; however, unless the time is reset *every* time, **make** will not work correctly.

Building a large program

As shown earlier, **make** can ease the task of generating a large program. The following is the **makefile** used to generate the screen editor MicroEMACS:

```
# MS-DOS limits command line tails to no more
# than 128 characters.  To skirt this limit, the
# command line is built into a temporary file,
# which we pass to make.

O1 = ansi.obj basic.obj buffer.obj display.obj file.obj \
        fileio.obj line.obj main.obj
O2 = window.obj word.obj tcap.obj
O3 = random.obj region.obj search.obj spawn.obj termio.obj vt52.obj

me.exe: $(O1) $(O2) $(O3)
        echo $(O1) > maketemp
        echo $(O2) >> maketemp
        echo $(O3) >> maketemp
        cc -o me.exe @maketemp
        del maketemp

$(O1) $(O2) $(O3): ed.h
```

This file shows how the elements of a **makefile** are used to control the generation of a large program.

The first four lines consist of comments that describe a peculiarity of the file, as fair warning to future programmers.

The next four lines define the macros **O1**, **O2**, and **O3**, which substitute for the 17 files that make up this program. Three macros must be used because, as explained in the comments, under MS-DOS no command line can have a tail longer than 128 characters.

The next line gives the name of the target file, **me.exe**, and the files needed to generate it; in this case, these file names are represented by the macros **O1**, **O2**, and **O3**.

The next three lines begin with the command **echo**. These command lines copy the three macros into the temporary file **maketemp**; this strategy is one way around the 128-character limit on command lines.

The next line is the command line. It controls the compiling of the files listed in **maketemp**.

The next to last line deletes **maketemp**, so that this file is no longer cluttering up your directory. Finally, the last line notes that all 17 of the MicroEMACS object modules are built in from the header file **ed.h**.

Command line options

Although **make** is controlled by your **makefile**, you can also control **make** by using command line options. These allow you to alter **make**'s activity without having to edit your **makefile**.

Options must follow the command name on the command line and begin with a hyphen, '-', using the following format. The square brackets merely indicate that you can select any of these options; do *not* type the brackets when you use the **make** command:

```
make [ -dinprst ] [ -f filename ]
```

Each option is described below.

-d (debug) **make** describes all of its decisions. You can use this to debug your **makefile**.

-f *filename*
(file) option tells **make** that its commands are in a file other than **makefile**. For example, the command

```
make -f smith
```

tells **make** to use the file **smith** rather than **makefile**. If you do not use this option, **make** searches the directories named in the environmental variable **PATH**, and then the current directory for a file entitled **makefile** to execute.

-i (ignore errors) **make** ignores error returns from commands and continues processing. Normally, **make** exits if a command returns an error status.

-n (no execution) **make** tests dependencies and modification times but does not execute commands. This option is especially helpful when constructing or debugging a **makefile**.

-p (print) **make** prints all macro definitions and target descriptions.

-r (rules) **make** does not use the default macros and commands from **$LIBPATH\mmacros** and **$LIBPATH\mactions**. These files will be described below.

-s (silent) **make** does not print each command line as it is executed.

-t (touch) **make** changes the modification time of each executable file and object module to the current time. This suppresses recreation of the executable file, and recompilation of the object modules. Although this option is used typically after a purely cosmetic change to a source module or after adding a definition to a header file, it must be used with great caution.

Other command line features

In addition to the options listed above, you may include other information on your command line.

First, you can define macros on the command line. A macro definition must *follow* any command line options. For example, the command line

```
make -n -f smith "CSD=-VCSD"
```

tells **make** to run in the *no execution* mode, reading the file **smith** instead of **makefile**, and defining

the macro **CSD** to mean **-VCSD**.

The ability to define macros on the command line means that you can create a **makefile** using macros that are not yet defined; this greatly increases **make**'s flexibility and makes it even more helpful in creating and debugging large programs. In the above example, you can define a command line as follows:

```
cc $(CSD) example.c
```

When you define the macro **CSD** on the command line, then the program is compiled using the **-VCSD** option, which creates an executable that can be debugged with **csd**, the Mark Williams C Source Debugger. If the macro is not set, however, then it is simply skipped when the command line is executed, and the program is compiled in the usual manner.

Another command-line feature is the ability to change the name of the *target file* on the command line. Normally, the target file is the executable file that you wish to create, although, as will be seen, it does not have to be. As will be discussed below, a **makefile** can name more than one target file. **make** normally assumes that the target is the first target file named in **makefile**. However, the command line may name one or more target files at the end of the line, after any options and any macro definitions.

To see how this works, recall the program **factor** described above. **factor** is generated out of the source modules **factor.c** and **atod.c**. The command

```
make atod.obj
```

with the **makefile** outlined above would produce the following **cc** command line:

```
cc -c atod.c
```

if the object module **atod.obj** does not exist or is outdated. Here, **make** compiles **atod.c** to create the target specified in the **make** command line, that is, **atod.obj**, but it does not create **factor**. This feature allows you to apply your **makefile** to only a portion of your program.

The use of special, or *alternative*, target files is discussed below.

Advanced make

This section describes some of **make**'s advanced features. For most of your work, you will not need these features; however, if you create an extremely complex program, you will find them most helpful.

Default rules

The operation of **make** is governed by a set of *default rules*. These rules were designed to simplify the compilation of a typical program; however, unusual tasks may require that you bypass or alter the default rules.

To begin, **make** uses information from the files **mmacros** and **mactions** to define default macros and compilation commands. **make** looks for these files in the directories named in the environmental variable **LIBPATH**. **make** uses the commands in **mmacros** and **mactions** whenever the **makefile** specifies no explicit regeneration commands. The command line option **-r** tells **make** not to use the macros and actions defined in **mmacros** and **mactions**.

As shown in earlier examples, **make** knows by default to generate the object module **atod.obj** from the source module **atod.c** with the command

```
cc -c atod.c
```

The macro **.SUFFIXES** defines the suffixes **make** knows about by default. Its definition in **mmacros** includes both the **.obj** and **.c** suffixes.

make's files **mmacros** and **mactions** use pre-defined macros to increase their scope and flexibility. These are as follows:

$< This stands for the name of the file or files that cause the action of a default rule. For example, if you altered the file **atod.c** and then invoked **make** to rebuild the executable file **factor.exe**, **$<** would then stand for **atod.c**.

$*** This stands for the name of the target of a default rule with its suffix removed. If it had been used in the above example, **$* would have stood for **atod**.

$<** and **$* work *only* with default rules; these macros will not work in a **makefile**.

$? This stands for the names of the files that cause the action and that are younger than the target file.

$@ This stands for the target name.

You can use the macros **$?** and **$@** in a **makefile**. For example, the following rule updates the file **factor** with the objects defined by macro **$(OBJ)** that are out of date:

```
factor: $(OBJ)
        cc -c $? -lm
```

mmacros also contains a default command that describes how to build additional kinds of files:

- **AS** and **ASFLAGS** call the *assembler* to assemble **.obj** files out of source modules written in assembly language rather than C.

You can change the default rules of **make** by changing them in **mactions** and changing the definition of any of the macros as given in **mmacros**.

Double-colon target lines

An alternative form of target line simplifies the task of maintaining libraries. This form uses the double colon "::" instead of a single colon ':' to separate the name of the target from those of the files on which it depends.

A target name can appear on only one single-colon target line, whereas it can appear on several double-colon target lines. The advantage of using the double-colon target lines is that **make** will remake the target by executing the commands (or its default commands) for the *first* such target line for which the target is older than a file on which it depends.

For example, for the program **factor.exe** described earlier, assume that two versions of the source modules **factor.c** and **atod.c** exist: **factora.c** plus **atoda.c**, and **factorb.c** plus **atodb.c** The **makefile** would appear as follows:

```
OBJ1 = factora.obj atoda.obj
OBJ2 = factorb.obj atodb.obj

factor.exe :: $(OBJ1)
        cc -c $(OBJ1) -lm

factor.exe :: $(OBJ2)
        cc -c $(OBJ2) -lm
```

This **makefile** tells **make** to do the following: (1) Check if either **factora.obj** or **atoda.obj** is younger than **factor.exe**. (2) If either one is, regenerate **factor.exe** using this version of these files. (3) If neither **factora.obj** nor **atoda.obj** is younger than **factor.exe**, then check to see if either **factorb.obj** or **atodb.obj** is younger than **factor.exe**. (4) If either of them is, then regenerate **factor.exe** using the youngest version of these files.

Let's C

This technique allows you to maintain multiple versions of source files in the same directory and selectively recompile the most recently updated version without having to edit your **makefile** or otherwise trick the system.

You cannot target a file in both a single-colon and a double-colon target line.

Alternative uses

make is a program that helps you construct complex things from a number of simpler things.

make usually is used to build complex C programs: the executable file is made from object modules, which are made from source modules and header files. However, **make** can be used to create any type of file that is constructed from one or more source modules. For example, an accountant can use **make** to generate monthly reports from daily inventories: all the accountant has to do is prepare a **makefile** that describes the dependencies (that is, the name of the monthly report they wish to create and the names of the daily inventories from which it is created), and the command required to generate the monthly report. Thereafter, to recreate the report, all the accountant has to do to generate a monthly report is type **make**.

In another example, the **makefile** can trigger program maintenance commands. For example, the target name **backup** might define commands to copy source modules to another directory; typing **make backup** saves a copy of the source modules. Similar uses include removing temporary files, building libraries, executing test suites, and printing listings. A **makefile** is a convenient place to keep all the commands used to maintain a program.

The following example shows a **makefile** that defines two special target files, **printall** and **printnew**, to be used with the source files for the program **factor.exe**.

```
# This makefile generates the program "factor.exe".
# "factor.exe" consists of the source modules "factor.c" and
# "atod.c".  It uses the standard mathematics library
# libm, but it requires no special header files.

OBJ = factor.obj atod.obj
SRC = factor.c atod.c

factor: $(OBJ)
        cc -o factor $(OBJ) -lm

# program to print all the updated source modules
# used to generate the program "factor.exe"


printall:
        pr $(SRC) | print /p
        echo junk > printall

printnew: $(OBJ)
        pr $? | print /p
        echo junk > printnew
```

In this instance, typing the command

```
        make printall
```

forces **make** to generate the target **printall** rather than the target **factor.exe**, which is the default as it appears first in the **makefile**. The **pr** and **print** commands are then used to print a listing of all files defined by **SRC**. The macro **OBJ** cannot be used with these commands because it would trigger the printing of the object files, which would not be of much use. The word **junk** is echoed into an empty file, **prnew**. This new file serves only to record the time the listing is printed. This tactic is performed in order to record the time that the listing was last generated so that **make** will know what files have been updated when you next use **printnew**.

Typing the command

```
make printnew
```

forces **make** to generate the target **printnew** rather than the default target **factor**. **printnew** prints only the files named in the macro **SRC** that have changed since any files were last printed.

Special targets

A few target names have special meanings to **make**. The name of each special target begins with '.' and contains upper-case letters.

The target name **.DEFAULT** defines the default commands **make** uses if it cannot find any other way to build a target. The special target **.IGNORE** in a **makefile** has the same effect as the **-i** command line option. Similarly, **.SILENT** has the same effect as the **-s** command line option.

Errors

make prints "*command* exited with status *n*" and exits if an executed *command* returns an error status. However, it ignores the error status and continues processing if the **makefile** command line begins with a hyphen '**-**' or if the **make** command line specifies the **-i** option.

make reports an error status and exits if the user interrupts it. It prints "**can't open** *file*" if it cannot find the specification *file*. It prints "**Target** *file* **is not defined**" or "**Don't know how to make** *target*" if it cannot find an appropriate *file* or commands to generate *target*. Other possible errors include syntax errors in the specification file, macro definition errors, and running out of space. The error messages **make** prints are generally self-explanatory; however, a table of error messages and brief descriptions of them are given in a later section of this manual.

Exit status

make returns a status of zero if it succeeds and -1 if an error occurs.

Where to go from here

make is summarized in the Lexicon. Look there for more information about how to use it with C programs.