This is the documentation of version 1.19. You may want the documentation of the stable version (2.0) or of the well tested 2.1 development snapshot or our homepage.

Tutorial on writing makefiles

A *makefile* is the set of instructions that you use to tell makepp how to build your program. Makepp can accept most makefiles written for the standard unix make, but if you're starting from scratch, it is often much simpler to use some of makepp's advanced features. This is an introduction for writing makefiles that are specific to makepp.

If you already know a lot about writing makefiles, you might want to at least peruse the later sections of this file because they show the preferred way to do things with makepp, which is often different from the traditional way to do it with make. Another source of examples and advice on writing makefiles for makepp is makepp_cookbook.

Building a program from its source files can be a complicated and time-consuming operation. The commands are too long to be typed in manually every time. However, a straightforward shell script is seldom used for compiling a program, because it's too time-consuming to recompile all modules when only one of them has changed.

However, it's too error-prone to allow a human to tell the computer which files need to be recompiled. Forgetting to recompile a file can mean hours of frustrating debugging. A reliable automatic tool is necessary for determining exactly which modules need recompilation.

Makepp (short for Make-plus-plus, or **make++**) is a tool for solving exactly this problem. It is an improvement on the **make** program, a standard tool that has been around for many years. It relies either on its own builtin knowledge (in very simple cases), or on a file called a *makefile* that contains a detailed recipe for building the program.

Usually, the input files are program source code, and the output files are executables, but makepp doesn't care what they are. You can use a makefile to control any kind of procedure where you need to selectively execute certain commands depending on which files have changed. You could, for example, use makepp to do data analysis, where your input files are raw data and analysis programs, and your output files are processed data or graphs or whatever. Makepp will figure out which of the processed data files need to be updated whenever some of the data files or analysis programs change. The examples in this introduction will assume you are building an executable program from source code, but you can do a lot more with makepp than just that if you use your imagination.

If your program consists of a single module, you probably don't need makepp, because you know that any change that you make requires recompiling that module. However, if your program consists of even just two modules, then you will definitely want to use a program like makepp.

Do I need a makefile?

If your program is relatively simple and doesn't require anything particularly special, makepp may already know how to build it without your explicitly giving instructions. For example, suppose you have a program in a single source file, called `test.c`. You can just type `makepp test` and your program will build like this:

```
% makepp test
makepp: Entering directory `/somewhere/or/other'
gcc -g -Wall -c test.c -o test.o
gcc -g -Wall test.o     -o test
Warning: on unix, to run a program called 'test', you must type
   ./test
rather than just 'test'.
```

These are the basic commands needed to compile a program on unix. If these commands don't make any sense to you, see makepp_tutorial_compilation.

Makepp contains builtin rules for C, C++, and Fortran.

Makepp can sometimes figure out how to compile programs that are contained in more than one source file, or programs that must be linked with various system libraries. It does this by guessing which source files and libraries you need based on the files that you include. The actual algorithm is too complicated to discuss here in a tutorial (but see makepp_builtin); you can try it, and if it doesn't work automatically for you, you need to write your own makefile.

By default, for C and C++, makepp compiles the program with debug information and without optimization. If you want to turn on optimization so that your program runs faster, change the command line to:

```
makepp CFLAGS=-O2 test
```

If you're compiling C++ instead of C, use `CXXFLAGS=-O2` instead of `CFLAGS=-O2`. For a complete list of other options you can configure without writing a makefile, see makepp_builtin.

Makepp's builtin rules are somewhat more powerful than the standard unix make, but if you write programs of any complexity, it's likely that you'll need a makefile eventually to tell makepp what to do.

If you are not familiar with unix compilation commands, it may be helpful at this point to read makepp_tutorial_compilation for a description of what these cryptic unix compilation commands do.

A simple makefile

Suppose you are writing a C++ program which has two source modules, `processing.cxx` and `gui.cxx`, along with numerous include files. If you were to build your program from scratch, you would need to execute something like these commands:

```
c++ -c processing.cxx -o processing.o
c++ -c gui.cxx -o gui.o
c++ processing.o gui.o -o my_program
```

The first two commands are compilation commands, and the third invokes the linker to combine the two object files into a single executable. If you make changes to `gui.cxx` but not to `processing.cxx`, then you don't need to reexecute the first command, but you do need to execute the last two commands. makepp can figure this out for you automatically.

(If you've never worked with make before, you may be thinking that you could combine the above three commands into a single command, like this:

```
c++ processing.cxx gui.cxx -o my_program
```

When you omit the `-c` option to the compiler, it combines the compilation and linking step. This is often quite convenient when you are not writing a makefile. However, it's not a good idea to do this in a makefile, because it always recompiles both modules even if one of them hasn't changed, and this can take a significant amount of extra time.)

In order to use makepp to control the build process, you'll need to write a *makefile*. The makefile is a text file that contains the recipe for building your program. It usually resides in the same directory as the sources, and it is usually called `Makefile`.

Each one of these commands should be a separate *rule* in a makefile. A rule is an instruction for building one or more output files from one or more input files. Makepp determines which rules need to be reexecuted by determining whether any of the input files for a rule have changed since the last time the rule was executed.

A rule has a syntax like this:

```
output_filenames : input_filenames
    actions
```

The first line of the rule contains a space-separated list of output files, followed by a colon, followed by a space-separated list of input files. The output files are also called *targets*, and the input files are also called *dependencies*; we say that the target file depends on the dependencies, because if any of the dependencies change, the target must be rebuilt.

The remaining lines of the rule (the *actions*) are shell commands to be executed. Each action must be indented with at least one space (traditional make requires a tab character). Usually, there's just one action line, but there can be as many as you want; each line is executed sequentially, and if any one of them fails, the remainder are not executed. The rule ends at the first line which is not indented.

You can place the rules in any order in the makefile, but it is traditional to write the rule that links the program first, followed by the compilation rules. One reason for this is that if you simply type “`makepp`”, then makepp attempts to build the first target in the file, which means that it will build your whole program

and not just a piece of it. (If you want to build something other than the first target, you have to specify the name of the target on the command line, e.g., "`makepp processing.o`".)

The above compilation commands should be written as three separate rules. A makefile for building this program could look like this:

```
# Link command:
my_program: processing.o gui.o
    c++ processing.o gui.o -o my_program

# Compilation commands:
processing.o: processing.cxx
    c++ -c processing.cxx -o processing.o

gui.o: gui.cxx
    c++ -c gui.cxx -o gui.o
```

(Characters on a line following a `#` are ignored; they are just comments. You do not need the "`# Link command:`" comment in the makefile at all.)

To use this makefile, simply cd to the directory and type "`makepp`". Makepp will attempt to build the first target in the makefile, which is `my_program`. (If you don't want it to build the first target, then you have to supply a the name of the target you actually want to build on the command line.)

When makepp attempts to build `my_program`, it realizes that it first must build `processing.o` and `gui.o` before it can execute the link command. So it looks at the other rules in the makefile to determine how to build these.

In order to build `processing.o`, makepp uses the second rule. Since `processing.o` depends on `processing.cxx`, makepp will also try to make `processing.cxx`. There is no rule to make `processing.cxx`; it must already exist.

Makepp checks whether `processing.cxx` has changed since the last time `processing.o` was built. By default, it determines this by looking at the dates on the file. Makepp remembers what the date of `processing.cxx` was the last time `processing.o` was made by storing it in a separate file (in a subdirectory called `.makepp`). Makepp will execute the actions to build the target if any of the following is true:

- The target does not exist.
- The target exists, but makepp does not have any information about the last build.
- The date on any input file has changed since the last build.
- The date on any target has changed since the last build.

- The actions have changed since the last build.
- The last build occured on a different architecture (different CPU type or operating system type).

It might seem a little funny that makepp executes the action if either the output file or the input files have changed since the last build. Makepp is designed to guarantee that your build is correct, according to the commands in the makefile. If you go and modify the file yourself, then makepp can't guarantee that the modified file is actually correct, so it insists on rebuilding. (For more information on how makepp decides whether to rebuild, and how you can control this, see makepp_signatures.)

Now `processing.o` might not depend only on `processing.cxx`; if `processing.cxx` includes any `.h` files, then it needs to be recompiled if any of those `.h` files has changed, even if `processing.cxx` itself has not changed. You could modify the rule like this:

```
# Unnecessary listing of .h files
processing.o: processing.cxx processing.h simple_vector.h list.h
    c++ -c processing.cxx -o processing.o
```

However, it is a real nuisance to modify the makefile every time you change the list of files that are included, and it is also extremely error prone. You would not only have to list the files that `processing.cxx` includes, but also all the files that those files include, etc. **You don't have to do this.** Makepp is smart enough to check for include files automatically. Any time it sees a command that looks like a C or C++ compilation (by looking at the first word of the action), it reads in the source files looking for `#include` directives. It knows where to look for include files by scanning for `-I` options on your compiler command line. Any files which are included are automatically added to the dependency list, and any files which those include. If any of them has changed, the file will be recompiled.

Once makepp knows that `processing.o` is up to date, it then determines whether `gui.o` needs to be rebuilt by applying the same procedure to the third rule. When both `processing.o` and `gui.o` are known to be built correctly, then makepp applies the same procedure to see if the link command needs to be reexecuted.

The above makefile will work, but even for this simple problem, an experienced user is not likely to write his makefile this way. Several improvements are discussed in the next sections.

Using variables

So far, our makefile for compiling our program of two modules looks like this:

```
# Link command:
```

```
my_program: processing.o gui.o
    c++ processing.o gui.o -o my_program

# Compilation commands:
processing.o: processing.cxx
    c++ -c processing.cxx -o processing.o

gui.o: gui.cxx
    c++ -c gui.cxx -o gui.o
```

This works wonderfully, but suppose now we want to change some compilation options. Or maybe we want to use a different compiler. We'd have to change all three compilation lines.

Similarly, suppose we want to change the list of modules to compile. We'd have to change it in two places.

Duplication of information like this is a recipe for disaster. If you go and change your makefile, it's pretty much guaranteed that at some point, you or someone else will forget to change one of the places. Depending on what the change is (especially if it affects preprocessor definitions), this can lead to subtle and hard-to-debug problems in your program.

The way to avoid duplication of information is to specify the information only once and store it in a variable, which can be accessed each time the information is needed.

```
# Define the symbols we might want to change:
CXX := c++
CXXFLAGS := -g

OBJECTS     := processing.o gui.o

my_program: $(OBJECTS)
    $(CXX) $(OBJECTS) -o my_program

processing.o: processing.cxx
    $(CXX) $(INCLUDES) $(CXXFLAGS) -c processing.cxx -o processing.o

gui.o: gui.cxx
    $(CXX) $(CXXFLAGS) -c gui.cxx -o gui.o
```

Here `$(CXX)` expands to be the value of the variable `CXX`, and similarly for `$(CXXFLAGS)` and `$(OBJECTS)`. Now we can just change one line in the makefile, and all relevant compilation commands are affected.

In fact, we don't even need to change the makefile to change compilation options. Assignments specified on the command line override assignments in

the makefile. For example, we could type this to the shell:

```
makepp CXXFLAGS="-g -O2"
```

which overrides the setting of `CXXFLAGS` in the makefile. It is as if the makefile contained the line

```
CXXFLAGS := -g -O2
```

instead of the definition it does contain.

It might not at all be useful to be able to override these things for your own development, but if you distribute your sources to other people, they might appreciate it.

Variable names are case sensitive (e.g., `OBJECTS` is different from `objects`). Usually people write most variables in upper case only, but you don't have to.

If you need to put a literal dollar sign into a rule action, write it with a double dollar sign, like this:

```
test:
    for testfile in *.test; do run_test $$testfile; done
```

Conventionally, there are a few variables which you might want to set. These are just conventions, but you will see them in a lot of makefiles.

```
CC      := cc              # The C compiler.
CFLAGS  := -g              # C compilation options which relate to
                           # optimization or debugging (usually
                           # just -g or -O).  Usually this wouldn't
                           # include -I options to specify the
                           # include directories, because then you
                           # couldn't override it on the command line
                           # easily as in the above example.
CXX     := c++             # The C++ compiler.  (Sometimes "CPP" instead
                           # of CXX.)
CXXFLAGS := -g             # C++ compilation options related to
                           # optimization or debugging (-O or -g).
F77     := f77             # The fortran compiler.
FFLAGS  :=                 # Optimization flags for fortran.
```

Makepp will guess appropriate values for some of these variables if you don't specify them (see makepp_builtin), but it is usually best to set them explicitly--it makes it easier on anyone reading your makefile.

There are a lot more extremely powerful things you can do with variables, but first we need to explain some more things about makefiles.

Pattern rules

Having one rule for each compilation command is fine when there are only a few files, but what if your program consists of dozens of source files? Most of them have to be compiled with very similar commands. It is tedious to type in a separate rule for each source file, and then if you decide to change the rules, you have to change the makefile in a dozen places. A better solution to this problem is to use a *pattern rule*.

A pattern rule is a concise way of specifying a rule for many files at once. The rule will depend on the file names, but usually it depends on them in a simple way. You specify a pattern by using the `%` wildcard. When present in the dependency list, `%` matches any string of any length; when present in the list of targets, `%` stands for the string that `%` in the dependency list matched.

The following pattern rule will take any `.c` file and compile it into a `.o` file:

```
%.o: %.c
    $(CC) $(CFLAGS) $(INCLUDES) -c $(input) -o $(output)
```

(This assumes that you have the variables `CC`, `CFLAGS`, and `INCLUDES` defined to be something suitable. Makepp will guess a value for `CC` and `CFLAGS`.)

The first line of the rule says that it applies to every possible input file that matches the pattern `%.c`. These `.c` files can be transformed into the corresponding `.o` file using the specified actions.

The action of rule is quite similar to the other actions we've seen previously, except that it uses *automatic variables*. An automatic variable is a variable whose value is automatically set by makepp depending on the rule that it appears in. The most useful automatic variables are:

$(input)
: The name of the first input file. In this rule, this would be the file that matches the `%.c` pattern. `$(dependency)` is a synonymn for `$(input)`. In older makefiles, you will also see the cryptic symbol `<` used as well.

$(output)
: The name of the first output file. In this rule, this would be the file that matches the `%.o` pattern. `$(target)` and `$@` are synonymns.

$(inputs)
: The name of all explicitly listed input files. In this case, since there is only one, `$(inputs)` is equivalent to `$(input)`. `$(dependencies)` and `$^` are synonymns.

$(outputs)

The name of all explicitly listed targets. In this case, since there is only one, $(outputs) is equivalent to $(output). $(targets) is a synonymn for $(outputs).

Note that these variables are lower case.

You can use these automatic variables even for non-pattern rules. This avoids repeating target filenames.

You can actually do considerably more complicated things with pattern rules. For example,

```
# Put the object files into a separate directory:
objects/%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $(input) -o $(output)

# Run a preprocessor to make source files:
moc_%.cxx: %.h
    $(MOC) $(input) -o $(output)
```

Using pattern rules and automatic variables, we'd probably rewrite our makefile for our simple program like this:

```
CXX := c++
CXXFLAGS := -g
INCLUDES := -I.             # This would contain any -I options to the
                            # compiler, if there are any.
LIBS     := -L/usr/X11R6/lib -lX11 # Contains libraries we need to link in.
OBJECTS  := processing.o gui.o

my_program: $(OBJECTS)
    $(CXX) $(inputs) -o $(output) $(LIBS)

%.o: %.cxx
    $(CXX) $(INCLUDES) $(CXXFLAGS) -c $(input) -o $(output)
```

Now we don't have to have an explicit rule for each object file we need to produce. If we want to add another module to our program, we only have to change the one line that defines the OBJECTS variable. Note that this makefile is now much more concise than our original makefile. Each piece of information occurs only once so there is no possibility of making a mistake by changing information in one place and forgetting to change it in others.

When you use pattern rules, it's not uncommon for there to be two different rules that can produce the same file. If both rules are pattern rules, then the one that occurs later in the makefile is actually used. If one rule is a pattern

rule, and the other is an explicit rule (one that actually names the target file explicitly), then the explicit rule is used. This is often helpful if you want to compile most modules with the same command, but there is one module that needs slightly different compilation options, as shown in this makefile fragment:

```
CXXFLAGS := -g -O2
FAST_CXXFLAGS := -DNO_DEBUG -O6 -malign-double -funroll-all-loops

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $(input) -o $(output)

time_critical_subs.o: time_critical_subs.cpp
    $(CXX) $(FAST_CXXFLAGS) -c $(input) -o $(output)
```

There is also another syntax that can be more convenient for affecting compilation options for just one or a few targets. It is possible to tell makepp that a variable should have a different value for certain specific targets. In this example, it would look like this:

```
CXXFLAGS := -g -O2
FAST_CXXFLAGS := -DNO_DEBUG -O6 -malign-double -funroll-all-loops

%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $(input) -o $(output)

time_critical_subs.o: CXXFLAGS := $(FAST_CXXFLAGS)
```

In general, if you specify a variable name after a list of targets, then it takes a different value when the build command for those targets is being determined.

If you find yourself wanting to do something with patterns that isn't expressed easily using the `%` wildcard, makepp has another syntax which is somewhat harder to read, but considerably more powerful. See the `:foreach` clause for more details.

Makepp actually has builtin rules for compiling C or C++ or Fortran code, which are available if you don't override them with your own rules. The builtin rules are almost identical to the examples above. Most makefiles contain pattern rules for compilation, but you can depend on the builtin rules if you want.

Phony targets

Often it is convenient to put commands into the makefile that don't actually build a file, but are somehow logically associated with the build process. For example, a very common procedure in makefiles is something like this:

```
prefix=/usr/local

install: our_program
    install -m 0755 our_program $(prefix)/bin
    install -m 0644 *.png $(prefix)/share/our_program/icons

.PHONY: install
```

When someone types `makepp install`, then makepp first builds `our_program`, then runs the commands associated with the install target. The `install` command simply copies its arguments to the specified directory, and sets the file's protection to the indicated value. So it copies `our_program` into `/usr/local/bin`, and some associated data files into `/usr/local/share/our_program/icons`. But this doesn't create a file called `install` in the current directory.

The `install` target here is called a *phony target* because makepp treats it as if it were a real file, but it is not actually a file, it's just a trick for forcing makepp to build its dependencies and then run some commands.

That's what the line .PHONY: install

is for. It tells makepp that it really shouldn't expect the file `./install` to exist after the commands have executed. If you forget the phony declaration, then makepp will expect the file `install` to exist after executing the commands, and it will complain loudly if it does not.

You can also write the phony declaration like this: $(phony install): our_program ...

and then omit the `.PHONY: install` line. This means that you can declare the target as phony on the same line as you define it, which may make your makefiles more readable.

Phony targets are extremely common in makefiles. In almost all makefiles, the first target is the phony target `all`, like this:

```
$(phony all): program1 program2 program3
```

If no target is specified on the command line, makepp attempts to build the first target in the file. If your makefile makes more than just one program, you most likely want to build all of the programs by default. In this example, if the programmer just types `makepp` without any arguments, makepp attempts to build `all`, which forces it to build all three programs from this directory.

Here is a sample makefile fragment that illustrates some commonly used phony targets:

```
PROGRAMS    := combobulator discombobulator

$(phony all): $(PROGRAMS)   # All is the first target, so it's the default.

combobulator: $(COMBOBULATOR_OBJS)
    $(CXX) $(inputs) -o $(output)

discombobulator: $(DISCOMBOBULATOR_OBJS)
    $(CXX) $(inputs) -o $(output)

#
# This target makes sure everything is compiled, and then puts the
# programs into a place where everyone can access them.  We make the
# directories if they don't exist yet.
#
prefix      := /usr/local

$(phony install): all
    test -d $(prefix) || mkdir $(prefix)
    test -d $(prefix)/bin || mkdir $(prefix)/bin
    for prog in $(PROGRAMS); do \
      install -m 0755 $$prog $(prefix)/bin; \
    done
    test -d $(prefix)/share || mkdir $(prefix)/share
    test -d $(prefix)/share/combobulate || mkdir -p $(prefix)/share/combobulate
    for icon in *.xbm; do \
      install -m 0644 $$icon $(prefix)/share/combobulate; \
    done
# Note the use of the double dollar sign to pass a single dollar sign to
# the shell.  Note also the backslashes at the end of a line to indicate
# that a shell command continues to the next line.

#
# This target gets rid of all the junk that gets built during compiles.
# (Note that this could be done more thoroughly with the only_targets
# function.)
#
$(phony clean):
    rm -f $(PROGRAMS) *.o

#
# This target makes a source distribution for shipping out to someone.
#
VERSION := 3.14

$(phony distribution):
    rm -rf combobulate-$(VERSION)   # Get rid of previous junk, if any.
    mkdir combobulate-$(VERSION)
    cp *.c *.h Makefile README INSTALL combobulate-$(VERSION)
    tar cf - combobulate-$(VERSION) | gzip -9c > combobulate-$(VERSION).tar.gz
    rm -rf combobulate-$(VERSION)

#
# This target runs regression tests to make sure the program(s) are
# doing what they are supposed to do.
```

```
#
$(phony test): $(PROGRAMS)
    noecho for testfile in *.test; do \
       ./combobulate $$testfile | ./discombobulate - > junk_output; \
      if cmp -s junk_output $$testfile; then \
        echo passed $$testfile; \
      else \
        echo failed $$testfile; \
      fi; \
    done
#
# If "noecho" is the first word of the action, the action itself is not
# printed before it is executed.  In this case, printing the action
# would merely clutter up the screen so it is very common to suppress
# printing for such long commands.
#
```

Working with several directories

If your program grows to a substantial size, or if it uses libraries that need to be built but should be kept separate, it is quite likely that you have split up your sources into several directories. One of the main motivations for writing makepp was to make dealing with several directories much easier than with the standard make utility. If you're familiar with the standard unix make, you'll notice that with makepp, you don't have to mess around with ugly complexities like recursive invocations of make.

With makepp, you simply put a separate makefile in each directory that builds the relevant files in that directory. When a makefile refers to files whose build commands are in different makefiles, makepp automatically finds the appropriate build rules in the other makefiles. All actions in each makefile are executed with the current directory set to be the directory containing the makefile, so each makefile can be written independently of all the others. No makefile has to know anything about the other makefiles; it does not even have to tell makepp to load the rules from those other makefiles.

When you've written your makefiles, cd to the directory that contains your main program, and type `makepp` just like you usually would. Makepp will load in the makefile from that directory. It will notice that this makefile refers to files in other directories, and it will examine those other directories to see if there is a makefile in them. In this way, all relevant makefiles will be loaded.

As a simple example, suppose your top level directory contains the following makefile:

```
# Top level makefile:
```

```
CXX := c++
CXXFLAGS := -O2
my_program: main.o goodies/libgoodies.so
    $(CXX) $(inputs) -o $(output)

%.o: %.cxx
    $(CXX) $(CXXFLAGS) -c $(input) -o $(output)
```

You would need to write a makefile in the directory `goodies` which builds `libgoodies.so`, like this:

```
# goodies/Makefile

CXX := c++
CXXFLAGS := -O2

MODULES = candy.o chips.o licorice.o cookies.o popcorn.o spinach.o

libgoodies.so: $(MODULES)
    $(CXX) -shared $(inputs) -o $(output)
                    # Note that the command is written assuming that
                    # the current directory is the subdirectory
                    # "goodies", not the top level subdirectory.
                    # Makepp cds into this directory before executing
                    # any commands from this makefile.

%.o: %.cxx
    $(CXX) $(CXXFLAGS) -fpic -c $(input) -o $(output)
```

And that's all you need to do.

Any variables which you specify on the command line override the definition of the variable in *all* makefiles. Thus, for example, if you type `makepp CXXFLAGS="-g"`, all modules will be recompiled for debug because the definition of `CXXFLAGS` in both makefiles is overridden.

The directories containing other sources need not be subdirectories of the top-level directory (as they are in this example). They can be anywhere in the file system; makepp will automatically load a makefile from any directory that contains a file which is a dependency of some target it is trying to build. It will also load a makefile from any directory that is scanned by a wildcard.

Automatic loading works if files built by your makefile all reside in the same directory as the makefile itself. If you write your makefile so that its rules produce files in a different directory than the makefile itself, then you might have to tell makepp where to look for the makefiles, since it doesn't have any way of guessing. You can do this using the `load_makefile` statement in your makefile. For more information about this and other issues related to multi-

directory builds, see makepp_cookbook/Tips for multiple directories.

One caveat: if you reference the variable `$(MAKE)` in your makefile, makepp automatically goes into backward compatibility mode and turns off automatic loading.

Template or boilerplate files

Makepp has several other features which make life slightly easier for programmers who have to maintain a program spanning several directories. In the above examples, you'll notice that the definitions of the variables `CXX` and `CXXFLAGS` have to be repeated in each makefile. It can be a nuisance to reenter the same information into every makefile, and it could be a problem if you ever decide to change it--you may have to modify dozens of different makefiles.

What you can do instead is to put all of the information that's common to each makefile into a separate file, located perhaps at the top of the directory tree. Common information usually includes variable definitions, and sometimes also pattern rules. (In the above example, however, the pattern rules are not the same in both makefiles.) Let's suppose you've called this file `standard_defs.mk`. Then each makefile simply needs to contain a statement like this:

```
include standard_defs.mk
```

When makepp sees this statement, it inserts the contents of the file into the makefile at that point. The `include` statement first looks for the file in the current directory, then in the parent of the current directory, and so on up to the top level of the file system, so you don't actually need to specify `../standard_defs.mk` or `../../../../standard_defs.mk`.

So we could rewrite the above makefiles to look like this. `standard_defs.mk` would exist in the top level directory, and it might contain the following definitions:

```
# standard_defs.mk
CXX := c++
CXXFLAGS := -O2

#
# We've also included a pattern rule that might be useful in one or more
# subdirectories.  This pattern rule is for C compilation for putting
# things into a shared library (that's what the -fpic option is for).
#
%.o: %.cxx
    $(CXX) $(CXXFLAGS) -fpic -c $(input) -o $(output)
```

Note that since the included file is actually inserted into each makefile, rules in

the included file are applied with the default directory set to the directory containing the makefile that included the file, not the directory containing the include file.

The top level `Makefile` might look like this:

```
# Top level makefile
include standard_defs.mk

my_program: main.o goodies/libgoodies.so
    $(CXX) $(inputs) -o $(output)

#
# Note that this pattern rule overrides the one found in standard_defs.mk,
# because makepp sees it later.  This pattern rule is for compilation for
# a module that doesn't belong in a shared library.
#
%.o: %.cxx
    $(CXX) $(CXXFLAGS) $(input) -o $(output)
```

And the subdirectory's makefile might look like this:

```
# goodies/Makefile
include standard_defs.mk

MODULES = candy.o chips.o licorice.o cookies.o popcorn.o spinach.o

libgoodies.so: $(MODULES)
    $(CXX) -shared $(inputs) -o $(output)

# We don't need the pattern rule for compilation of .cxx to .o files,
# because it's contained in standard_defs.mk.
```

The -F compilation option

If you run makepp from within an editor such as emacs, and you are editing sources from several different directories, you may find that the default directory for makepp differs depending on which file you were most recently editing. As a result, makepp may not load the correct makefile.

What you can do to ensure that makepp always loads the correct makefile(s), no matter what directory happens to be your current directory, is to use the `-F` command line option, like this:

```
makepp -F ~/src/my_program
```

Makepp will first cd to the directory `~/src/my_program` before it attempts to load a

makefile.

Using Wildcards

Up until this point, we've had to explicitly list all of the modules that go into a program or a library. The previous makefile, for example, contained this line:

```
MODULES = candy.o chips.o licorice.o cookies.o popcorn.o spinach.o

libgoodies.so: $(MODULES)
    $(CXX) -shared $(inputs) -o $(output)
```

In this case, listing all of the modules that go into `libgoodies.so` is not such a big deal since there aren't very many of them. But sometimes it can be a real nuisance to list all of the object files, especially if this list is changing rapidly during development. Frequently, you want every single module in the whole directory to be compiled into your program or library. It would be a lot easier if you could just tell makepp to do that without listing them all.

Well, you can. The above lines could be rewritten as:

```
libgoodies.so: *.o
    $(CXX) -shared $(inputs) -o $(output)
```

The `*.o` wildcard matches any existing `.o` files, or any `.o` files which do not yet exist but can be made by any of the rules that makepp knows about from any makefiles that it has read. So the wildcard will return the same list of files, no matter whether you haven't compiled anything yet, or whether all the modules have been compiled before.

Of course, if you contaminate your directories with extra files that shouldn't be compiled directly into your library, (e.g., if you write little test programs and leave them in same directory as the library source files), then these modules will be incorrectly included into your library. If you choose to use wildcards, it's up to you to keep the directory clean enough.

Makepp supports the usual unix wildcards and one additional one:

- Matches any string of 0 or more characters. It will not match the `/` character. For example, `a*c` matches `ac`, `abc`, and `aaaaabc`, but not `aa/bc`.
- Matches exactly one character (not including `/`). For example, `???.o` matches all filenames that have 3 characters before the `.o` extension.
- Matches any of a list of characters at that position. For example, `[abc].o` matches `a.o`, `b.o`, `c.o`, but not `abc.o` or `d.o`. You can also specify a range of

characters, e.g., `data_[0-9]` will match `data_0`, `data_1`, etc.
- This is a special wildcard, found only in makepp (and the zsh shell, from which I stole the idea). It matches any number of intervening directories. For example, `**/*.o` matches `xyz.o`, `test_programs/abc.o`, and `a/deeply/nested/subdirectory/def.o`.

  If your sources are contained in several subdirectories, and you want to link all the object modules together, you could write it like this:

```
liboodles.so: **/*.o
    $(CXX) -shared $(inputs) -o $(output)
```

Functions and Advanced Variable Usage

Makepp has a number of extremely powerful ways of manipulating text. This tutorial shows a few of the more useful ways, but you might want to glance through makepp_variables and makepp_functions for a more complete list.

Lists of corresponding files

A common problem in makefiles is the maintenance of two lists of files which correspond. Consider the following two variables:

```
SOURCES := a.cpp bc.cpp def.cpp
OBJS := a.o bc.o def.o
```

We might want to have a list of sources if the makefile can build source distributions, and we might need a list of objects for the link command. It's tedious to change both lines whenever a new module is added, and it's not unlikely that a programmer will change one line and forget to change the other. Here we will show four different ways to avoid the duplication.

The patsubst function

The first is to use makepp's functions to convert one list into another. A function invocation looks a little like a variable, except that a function can take arguments:

```
$(function arg1 arg2 arg3 ...)
```

Makepp supplies many powerful functions, but probably the most useful of them is the patsubst function. You could write the above lines like this:

```
SOURCES = a.cpp bc.cpp def.cpp
OBJS = $(patsubst %.cpp, %.o, $(SOURCES))
```

The patsubst function applies a pattern to every word in a list of words, and performs a simple textual substitution. Any words in the list that match the pattern in the first argument are put into the output after making the substitution indicated by the second argument. The % wildcard matches any string of 0 or more characters. In this example, the pattern %.cpp is applied to every word in $(SOURCES). The first word, a.cpp matches the pattern, and the % wildcard matches the string a. The % in the second argument is then replaced by a, and the result is a.o. For the second argument, % matches bc, so the result is bc.o.

Makepp's functions can strip directory names, remove extensions, filter out matching words, return the output from shell commands, and other useful tricks. In addition, you can also write your own functions in perl that can be called from other parts of the makefile. See makepp_extending for details.

Substitution references

Since the patsubst function is so common, there is an abbreviated syntax for it called a *substitution reference*. We could have written the above lines like this:

```
SOURCES = a.cpp bc.cpp def.cpp
OBJS = $(SOURCES:%.cpp=%.o)
```

rc-style substitution

Sometimes invocations of patsubst or the equivalent substitution references can be somewhat cryptic. Makepp provides another option which is sometimes more convenient: i<rc-style substitution (so called because it was pioneered by the rc shell).

```
MODULES := a bc def
SOURCES := $(MODULES).cpp
OBJS := $(MODULES).o
```

What happened here is that when it evaluated $(MODULES).cpp, makepp appended .cpp to every word in $(MODULES), and similarly for $(MODULES).o. In general, any characters preceding the $(variable) (up to a word delimiter) are placed before each word in $(variable), and any characters following $(variable) are placed after each word in $(variable). Thus the result of evaluating x$(MODULES)y would be xay xbcy xdefy.

Inline perl code

If you know perl, you can insert perl code to perform arbitrary

manipulations on variables into your makefile. This is best illustrated by an example:

```
SOURCES := a.cpp bc.cpp def.cpp
perl_begin
($OBJS = $SOURCES) =~ s/\.cpp/.o/g;
perl_end
```

Any text between the perl_begin statement and the perl_end statement is passed off to the perl interpreter. All variables in the makefile (except automatic variables) are accessible as perl scalars. Any variables you set with perl code will be accessible in the makefile.

So what the above example does is to copy the text from $SOURCES to $OBJS, then substitute each occurence of .cpp with .o.

In this example, using inline perl code is probably unnecessary since there are easier and clearer ways of doing the same manipulation. But the full power of the perl interpreter is available if you need it.

Source/Object Separation and Variant Builds

Up to this point all of the makefiles we have seen put the object files in the same directory as the source files. This is usually the way makefiles are written, and it's certainly the simplest way to do things. However, suppose you have to compile your program on both a linux machine and a Solaris machine. The binaries from the two machines are incompatible, of course. Unlike the traditional make, makepp is smart enough to know that if the last compilation was on linux, and the current compilation is on Solaris, a recompilation of everything is necessary.

But this still leaves a problem: when you recompile on Solaris, you wipe out your linux binaries. Then when you switch back to linux, you have to recompile everything again, even though the source files that haven't changed.

A related problem is if you build your program with several different options. Suppose for example that you usually compile your program with optimization:

```
CFLAGS      := -O2

%.o: %.c
    $(CC) $(CFLAGS) -c $(input) -o $(output)

my_program: *.o
    $(CC) $(inputs) -o $(output)
```

However, you discover a bug, and you want to enable debugging on all files, so you do change CFLAGS:

```
CFLAGS      := -g -DMALLOC_DEBUG
```

Makepp realizes that the build commands have changed, and it needs to recompile everything. But again, recompiling with debugging enabled wipes out your old binaries, so if you want to turn optimization back on, everything must be recompiled again, even the files that haven't changed.

The obvious solution to these problems is to put the architecture-dependent or build-variant-dependent files in a separate subdirectory. There are two basic techniques for doing this: explicitly specifying an alternate directory, or using repositories.

Explicit specifications of alternate directories

You could rewrite the rules in your makefile to dump the objects into a different directory, like this:

```
ARCH        := $(shell uname -m)    # ARCH becomes the output from the uname -m command.
CFLAGS      := -O2
OBJDIR      := $(ARCH)-optim

$(OBJDIR)/%.o: %.c
    $(CC) $(CFLAGS) -c $(input) -o $(output)

$(OBJDIR)/my_program: $(OBJDIR)/*.o
    $(CC) $(inputs) -o $(output)
```

Now when you run makepp, ARCH is automatically set to something different for each architecture, and all of the objects are placed in a different directory for each architecture, so they don't overwrite each other. If you want to recompile turning on debugging, then you would have to change both CFLAGS and OBJDIR.

One problem with this approach is that implicit loading will no longer work. The only place that makepp knows to look for a makefile when it needs to build something is in the directory of the file it's trying to build. If this is a problem for you, then you can explicitly tell makepp where to look using the load_makefile statement.

Repositories

Repositories are a magical way of using a makefile that is written to put objects in the same directory, but having makepp automatically put the objects in a

different directory. Suppose we start with the original makefile above (before we modified it to put the objects in a different directory), and we've been working on linux so our source directory is filled with linux binaries. When we want to recompile our code on solaris instead of linux, we use the following command instead of just typing makepp:

```
% mkdir solaris
% cd solaris
% makepp -R ..
```

What the `-R` option to makepp does in this case is to declare the directory `..` (which is the original source directory) as a repository. A repository is just a way of getting makepp to trick all of the actions into believing that all files in one directory tree are actually located in a different directory tree in the file system. In the above example, makepp pretends that all the files in `..` (and all subdirectories of `..`) are actually in the current directory (and corresponding subdirectories).

More precisely, a repository is a place where makepp looks if it needs to find a file that doesn't exist in the current directory tree. If the file exists in the current directory tree, it is used; if it doesn't exist, but a file exists in the repository, makepp makes a temporary symbolic link from the file in the repository to the current directory. (A symbolic link is an alias for the original file. It's like a copy, except that trying to access the link actually accesses the original file.) The rule actions then act on the file in the current directory, but actually reference the files in the repository.

In this example, initially we start off with a blank new directory `solaris`. (It doesn't have to be blank, of course, and it won't be the second time you run makepp.) Makepp is run in this directory, and it sees that there is no makefile there. However, there is a makefile in the repository, so it links in the one from the repository, and reads it. The pattern rule in the makefile that converts `.c` files into `.o` files causes makepp to link all the `.c` files that it needs from the repository, and run the compilation command from the `solaris` subdirectory. Therefore the `.o` files are now placed into the `solaris` subdirectory, not in the top level directory. When the build command is finished, any files linked from the repository are deleted, so the `solaris` subdirectory will contain only the binary files for Solaris. Any `.o` files that exist in the repository are unmodified, so when you go back to your linux machine and rerun makepp, most of your program will not have to be recompiled.

Sometimes it might be more convenient to use a different form of the repository command. The above three shell commands could be entirely replaced by the following one command:

```
% makepp -R solaris=. -F solaris
```

What this does is to say that the files in the current directory are to be linked into the `solaris` subdirectory as necessary. (The `solaris` subdirectory will be created automatically if it does not exist.) Then the `-F` option causes makepp to cd to the solaris directory and execute the makefile there (which will be linked from the repository).

Using a repository does not have the same drawbacks as explicitly specifying an object directory; makefiles will be implicitly loaded as expected, since as far as makepp is concerned, the makefile actually is in the same directory as the target files. However, if your build involves not just one but several directory trees, using repositories can become quite complicated.

Repositories are just a way of pretending that things located at one place in the file system are actually in a different place for the duration of the build. This is a very powerful technique that can be used for more than just separating your sources and binaries. For more details, see makepp_repositories.

Debugging Makefiles

.makepp_log

If you have a complicated build procedure, you find that makepp is rebuilding things more often than you think they need to be rebuilt. Or you may find that it is not rebuilding things when it should. You don't have to keep staring at your makefiles until you see the problem. On every build, makepp produces a log file that explains which rule it thought it was supposed to use to build each target, what files it thought each target depended on, and (if it did decide to rebuild) why it thought a rebuild was necessary. This file is called `.makepp_log` and it is placed in the directory you actually ran makepp from. (Of course, the filename has a leading period, which means that you won't see it there unless you specifically look for it. It's designed to be unobtrusive.)

The log file's format is more or less self-explanatory. Indentation in the log file conveys depth in makepp's inference tree. Suppose the target is `all`, and `all` depends on `my_program`, and `my_program` depends on `*.o`, which depend on the corresponding `.c` files. Log messages related to `all` will not be indented, log messages related to building the target `my_program` will be indented two spaces, log messages related to building any of the object files will be indented 4 spaces, and log messages related to building any of the source files will be indented 6 spaces.

If you're doing a parallel make (using the `-j` command line option), the order of the messages in the log file will not make nearly as much sense since messages from different targets will be interspersed. You might try debugging a serial make first.

Common errors in makefiles

Not specifying all dependencies

Makepp is designed to be extremely clever about finding dependencies, and if you just use a standard unix C or C++ compiler command, it is actually somewhat difficult to get makepp to miss something. (Please send me examples if you find that it missed something, so I can make makepp smarter.) However, if you are running commands other than compilation, or dealing with languages other than C or C++, it is much easier to run into problems.

If you don't tell makepp all of the dependencies of a file, and it can't infer them by looking at the command line or scanning the files for includes, then it may not rebuild a file when it should. You can make this kind of error less likely by using only automatic variables in your actions, rather than repeating the dependency lists. For example,

```
combined_file: a b c
     do_something a b c d > combined_file
```

has an error because *d* is mentioned in the action but not in the dependency list. If the command had been written using automatic variables like this:

```
combined_file a b c d
    do_something $(inputs) > combined_file
```

then it would have been impossible to make this mistake.

Another way that a missing dependency can occur is if a program actually uses a file but doesn't take the file's name on the command line. For example, if you're compiling Fortran code, makepp at the moment doesn't know how to scan for included files. Thus you must explicitly list any files that are included.

One thing that is sometimes helpful for testing is to start with a completely clean directory--just the bare minimum you think should be necessary--and rebuild absolute everything from scratch. This can be most conveniently done by using repositories, like this:

```
rm -rf test-build-dir
makepp -R test-build-dir=. -F test-build-dir
```

If the build fails because some file is not present, it means that makepp didn't realize some file was a dependency, because it only links files from the repository that it thought were needed. Performing this test occasionally may save hours of debugging later. I have worked on projects where this was never done for months because recompilation took so long. As a result, many little problems crept in. There were some object files that didn't have source files any more, some source files that were never properly rebuilt by a preprocessing command, etc.

Of course, this won't catch all missing dependencies, but it will catch some of them.

Not specifying all targets

You must specify **all** files that a given command modifies as targets, or else makepp may not have realized they have changed. You can specify more than one target. For example,

```
y.tab.h y.tab.c: parse.y
    yacc -d parse.y
```

If you had forgotten to specify *y.tab.h* as a target, then makepp would not know to rebuild *y.tab.h* using this command, and files that depend on *y.tab.h* might not be recompiled after the yacc command is run.

Please suggest things that you have found confusing or dangerous, and I'll either note them or try to fix makepp so they aren't a danger any more.

<!DOCTYPE HTML PUBLIC “-//IETF//DTD HTML//EN" <html <head <titleUsing Qt's moc preprocessor</title </head

<body <h1Using Qt's moc preprocessor</h1

<hr <a href="t_index.html"Tutorial index</a | <a href="t_debugging.html"Next (debugging makefiles)</a | <a href="t_functions.html"Previous (functions and variables)</a <br <!-- $Id: makepp_tutorial.pod,v 1.3 2003/07/15 00:50:38 grholt Exp $ -- <!-- Created: Fri Aug 25 23:04:23 PDT 2000 -- <!-- hhmts start -- Last modified: Tue Dec 26 21:08:53 PST 2000 <!-- hhmts end -- </body </html <!DOCTYPE HTML PUBLIC “-//IETF//DTD HTML//EN" <html <head <titleWhat is in a makefile</title </head

<body <h1What is in a makefile</h1

<hr <a href="t_index.html"Tutorial index</a | <a href="t_functions.html"Next (functions and variables)</a | <a href="t_dirs.html"Previous (directories)</a <br <!-- $Id: makepp_tutorial.pod,v 1.3 2003/07/15 00:50:38 grholt Exp $ -- <!-- Created: Fri Aug 25 21:16:52 PDT 2000 -- <!-- hhmts start -- Last modified: Fri Aug 25 22:57:09 PDT 2000 <!-- hhmts end -- </body </html <h1Libraries and Makepp</h1

Libraries are used for several reasons. <ol <liA subroutine or collection of subroutines that is used in many different programs is often put into a library. For example, the C functions printf, fprintf, etc., are all part of the standard C library that every program links with. <liA library

Two kinds of libraries are common on unix systems: static and dynamic (or shared). Static libraries have a `.a` suffix, and dynamic libraries have a `.so` (or sometimes `.sa`) suffix.