Linux Software Development Tools

Prof. Prabhat Ranjan

INSTALLING SOFTWARE PACKAGES

Many software packages come as compressed archives of source files.

The same package may be "built" to run on different target machines

A single distribution of a software package may thus end up running, in various incarnations, on an Intel box, a DEC Alpha, a RISC workstation, or even a mainframe.

Unfortunately, this puts the responsibility of actually "building" and installing the software on the end user

Unpacking the Files

downloaded software package is most likely in archived (tarred) and compressed (gzipped), in .tar.gz or .tgz form (familiarly known as a "tarball").

First copy it to a working directory.

Then untar and gunzip it.

The appropriate command for this is

```
tar xzvf <filename>
```

The de-archiving process will usually install the appropriate files in subdirectories it will create.

preview this process by a

```
tar tzvf <filename>
```

Source files in the new bzip2 (.bz2) format can be unarchived by a

bunzip2 <filename>

Sometimes the archived file must be untarred and installed from the user's home directory, or perhaps in a certain other directory, such as /, /usr/src, or /opt, as specified in the package's config info.

Should you get an error message attempting to untar it, this may be the reason.

Read the package docs, especially the README and/or Install files

Most software packages permit automating this process by running make install to place the binaries in the appropriate system areas.

You might encounter shar files, or shell archives, especially in the source code newsgroups on the Internet.

Occasionally, you may need to update or incorporate bug fixes into the unarchived source files using a patch or diff file that lists the changes.

The doc files and/or README file will inform you should this be the case.

The normal syntax for powerful patch utility is

```
patch < patchfile.
```

Creating Patch File

```
$ cat file1.0
Tis is line 1
This is line 2
This is line 3
This is line 4
```

```
$ cat file1.01
This is line 1
This is line 2
This is modified line 3
This is line 4
```

diff -Naur file1.0 file1.01

```
$ diff -Naur file1.0 file1.01
--- file1.0     2004-08-16 07:20:56.000000000 +0530
+++ file1.01    2004-08-16 07:21:24.000000000 +0530
@@ -1,4 +1,4 @@
-Tis is line 1
+This is line 1
 This is line 2
-This is line 3
+This is modified line 3
 This is line 4
```

Using Make

The Makefile is the key to the build process.

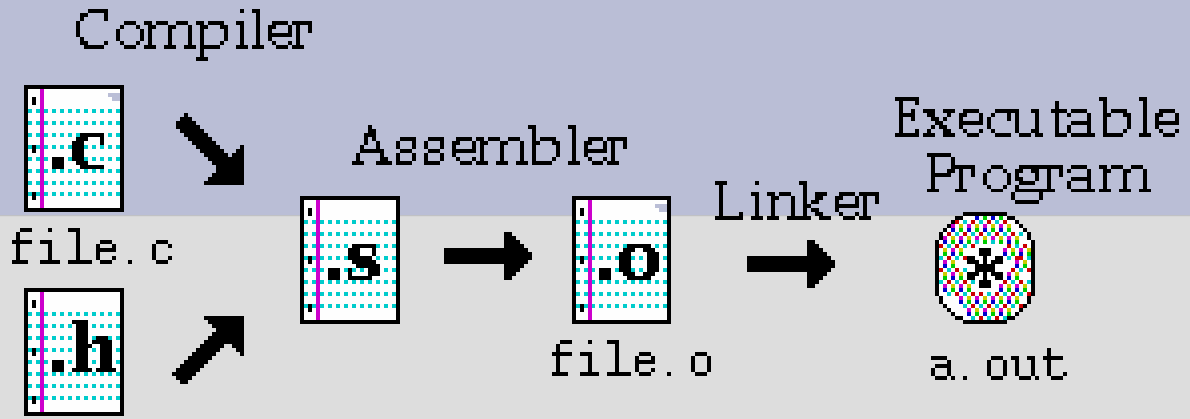
In its simplest form, a Makefile is a script for compiling or building the "binaries", the executable portions of a package.

The Makefile can also provide a means of updating a software package without having to recompile every single source file in it

At some point, the Makefile launches cc or gcc. This is actually a preprocessor, a C (or C++) compiler, and a linker, invoked in that order.

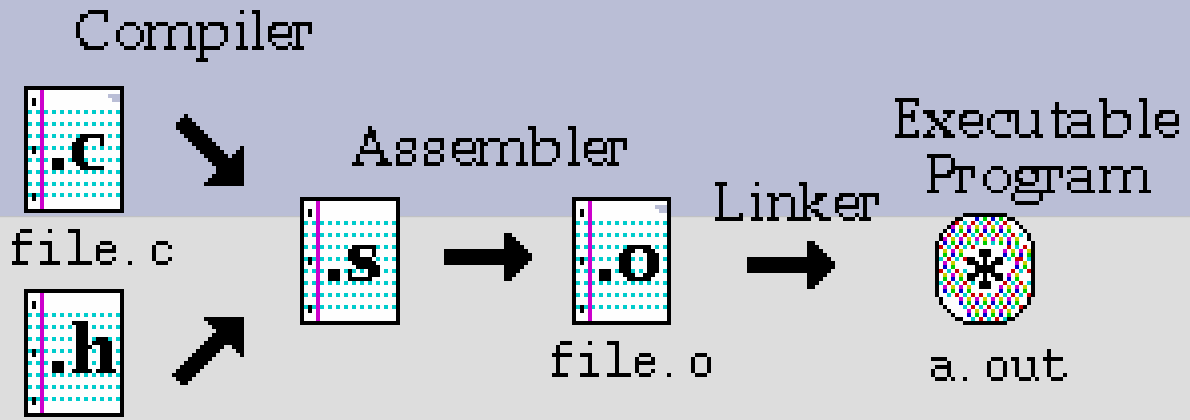
This process converts the source into the binaries, the actual executables.

A simple compilation

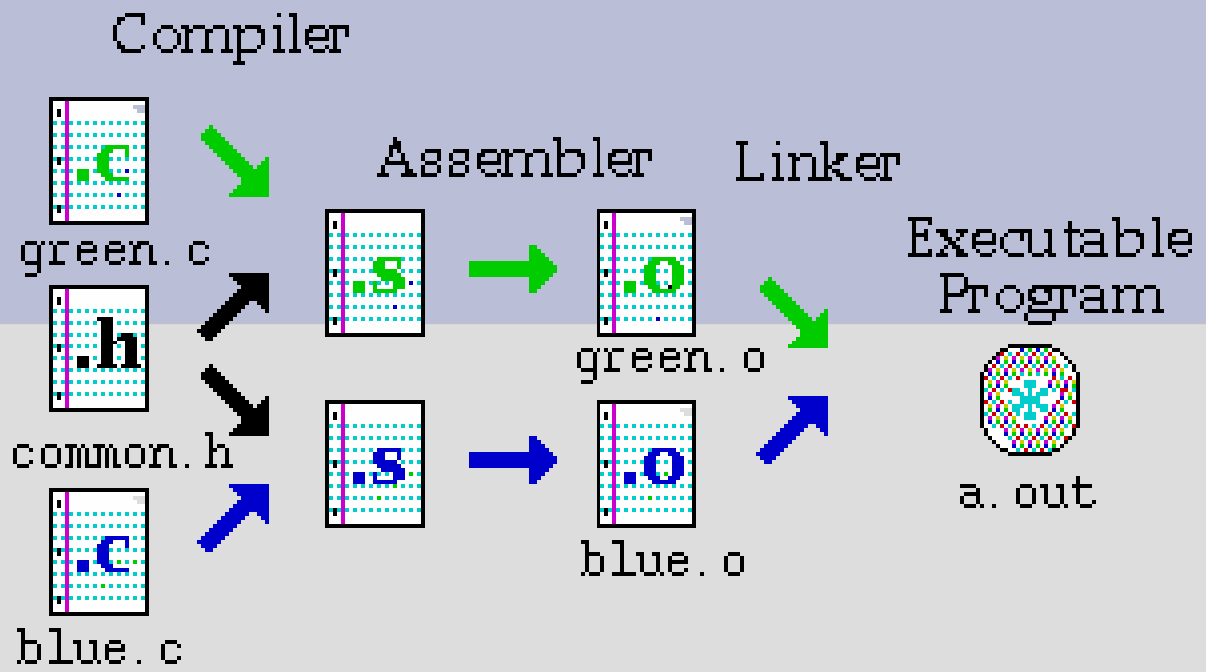


Compiling a small C program requires at least a single .c file, with .h files as appropriate. There are 3 steps to obtain the final executable program, as shown:

1. Compiler stage: All C language code in the .c file is converted into a lower-level language called Assembly language; making .s files.
2. Assembler stage: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file ends with .o.



3. Linker stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions, such as printf. This stage produces an executable program, which is named a.out by default.



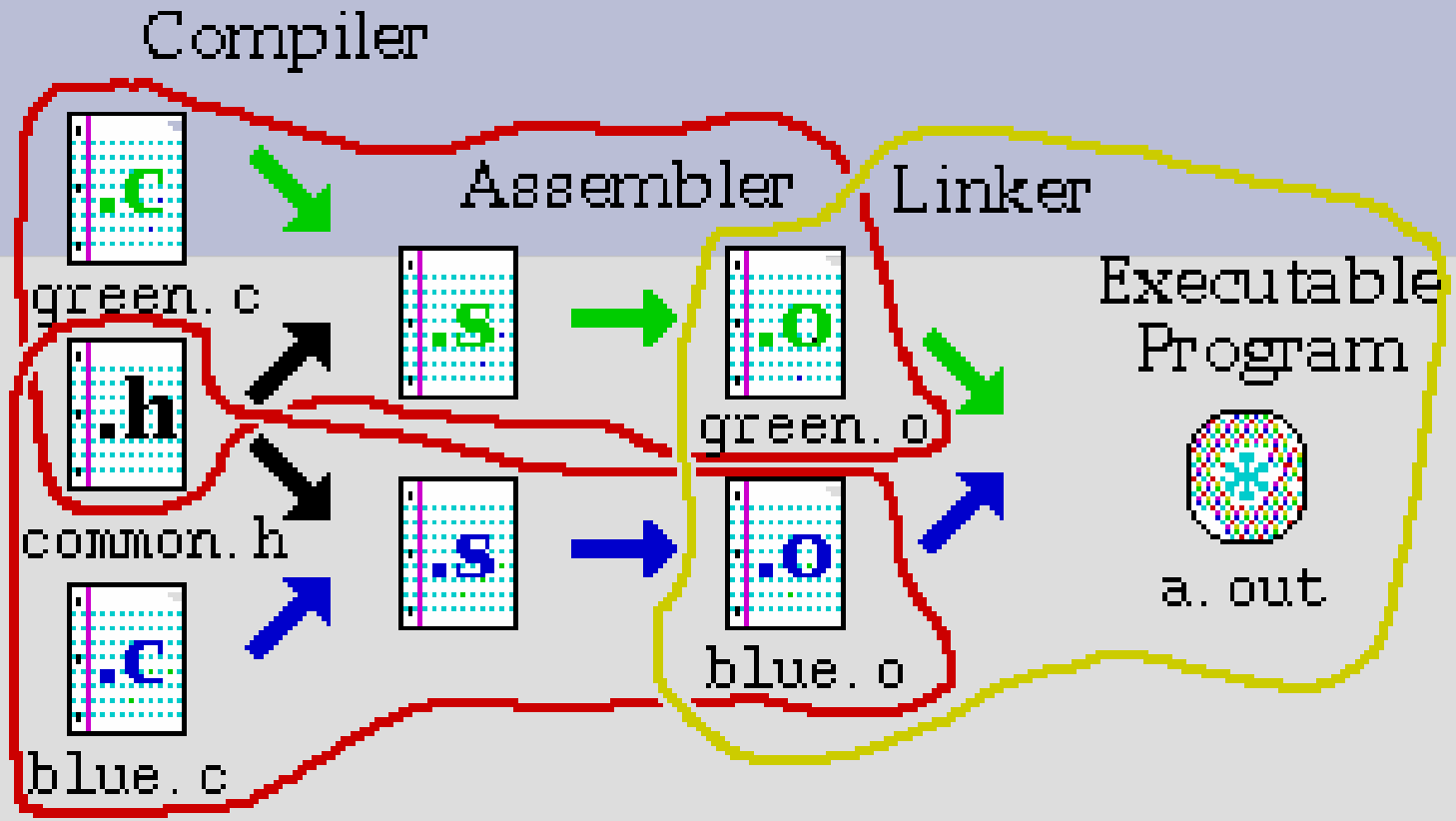
When program becomes very large, it makes sense to divide source code into separate easily-manageable .c files.

The command is as follows:

cc green.c blue.c

where both .c files are given to the compiler.

Separate compilation



The three different tasks required to produce the executable program are as follows:

* Compile green.o: cc -c green.c
* Compile blue.o: cc -c blue.c
* Link the parts together: cc green.o blue.o

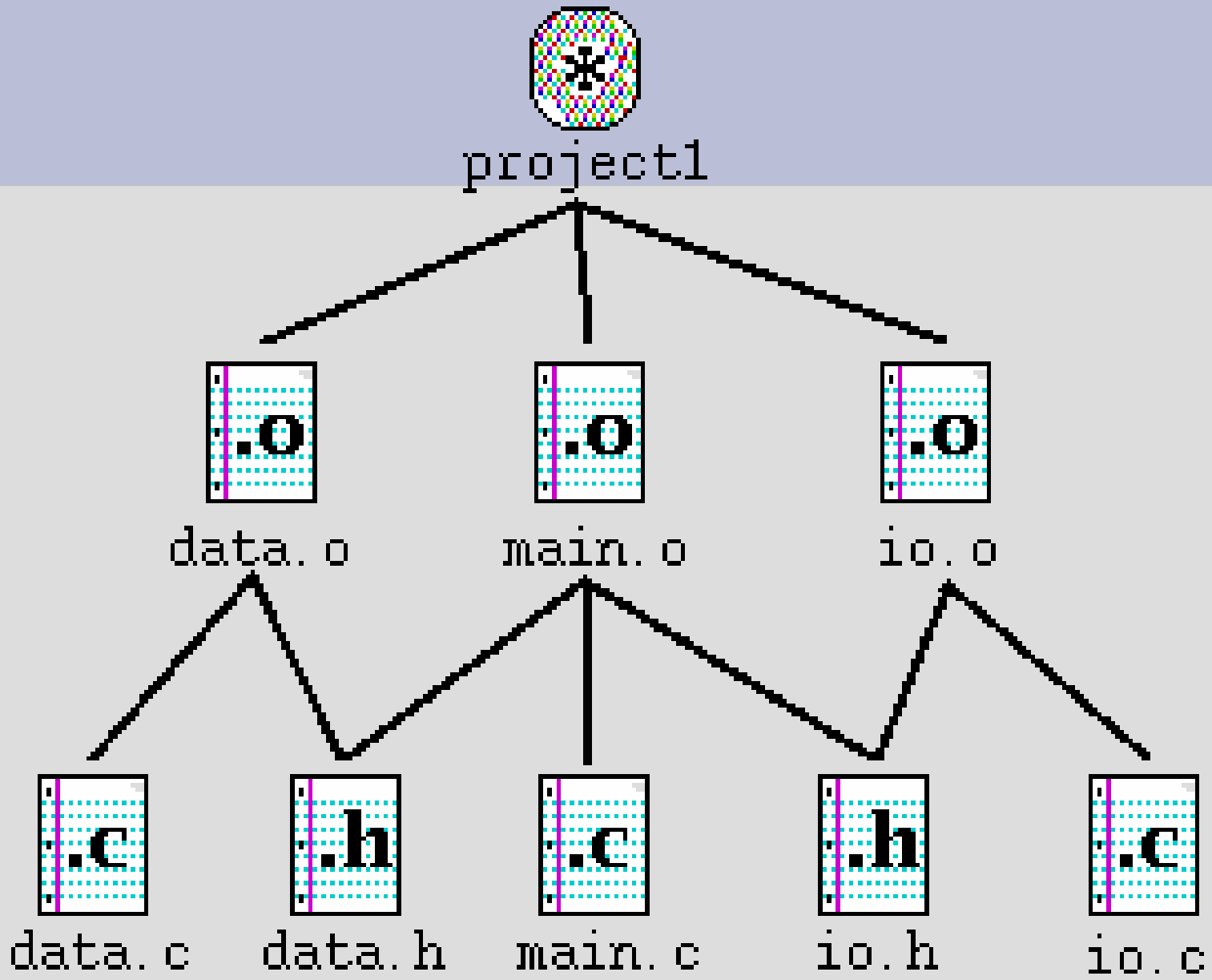
Dependencies

”Make” creates programs according to the file dependencies.

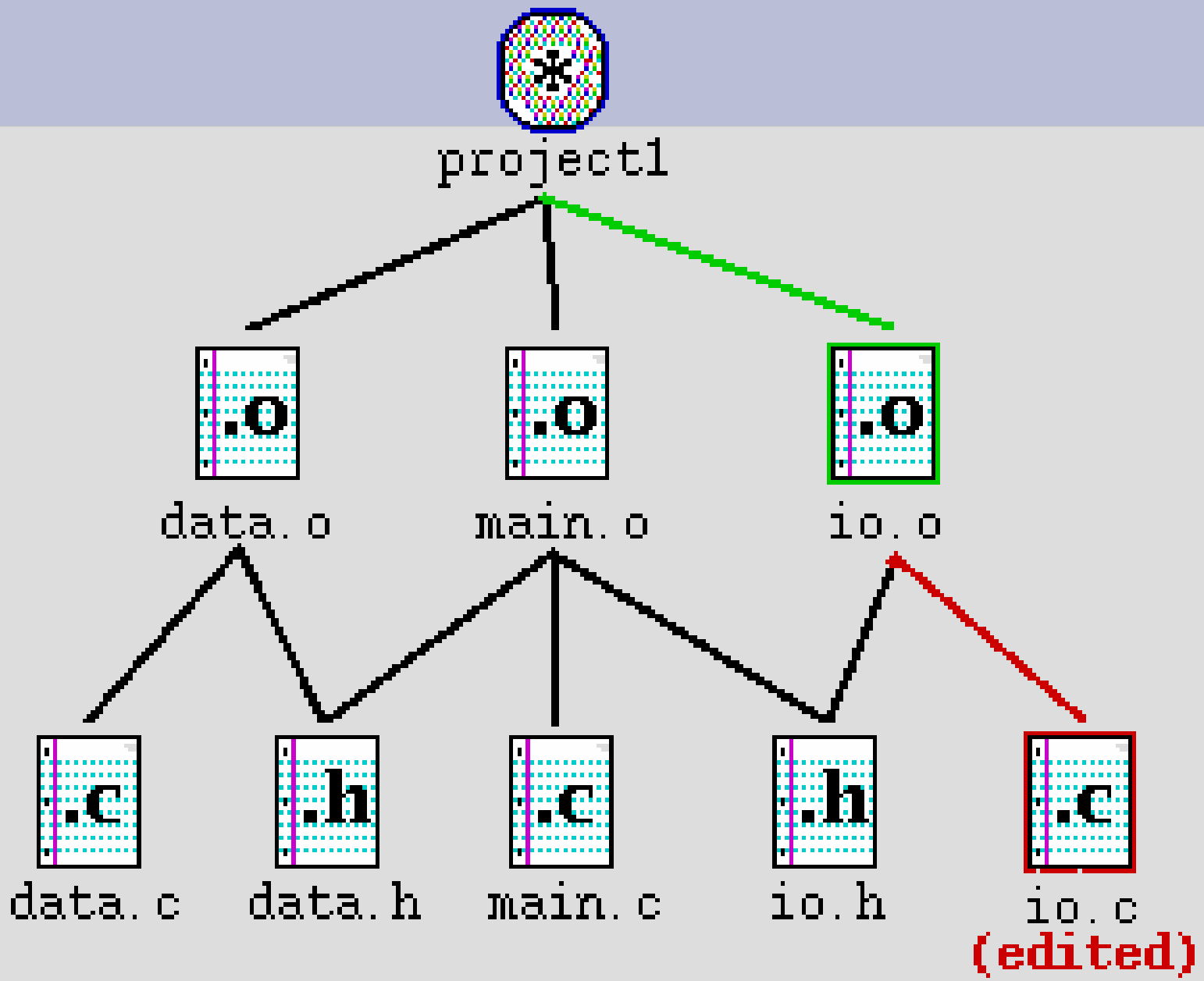
For example, in order to create an object file, program.o, we require at least the file program.c. (There may be other dependencies, such as a .h file.)

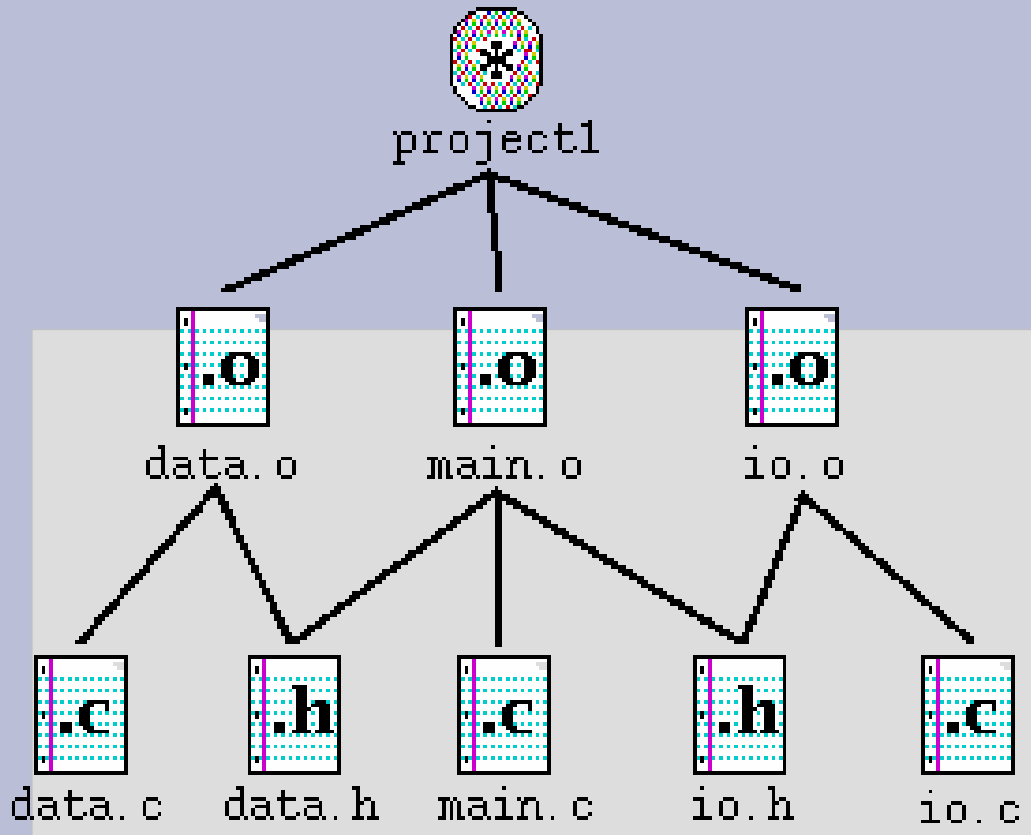
First step involves drawing what are called "dependency graphs", which are very similar to the diagrams given in the previous slide.

Dependency graphs



How dependency works

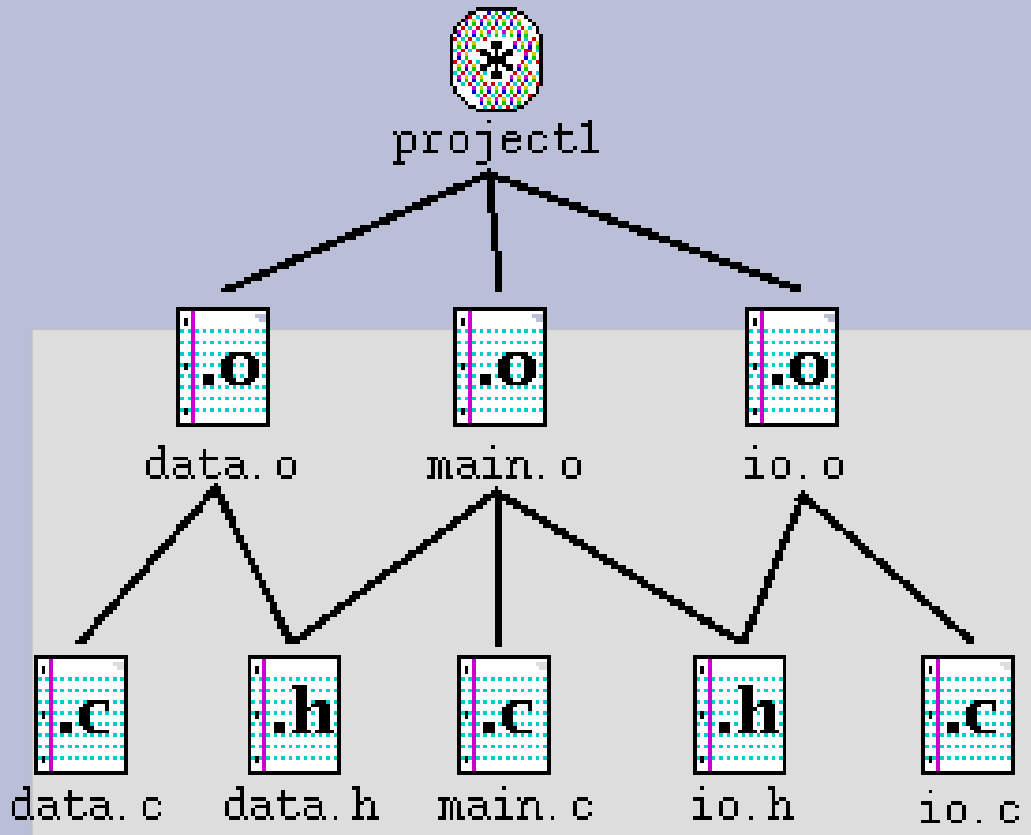




Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

The make program gets its dependency "graph" from a text file called makefile or Makefile which resides in the same directory as the source files.
Make checks the modification times of the files, and whenever a file becomes "newer" than something that depends on it, (in other words, modified) it runs the compiler accordingly.

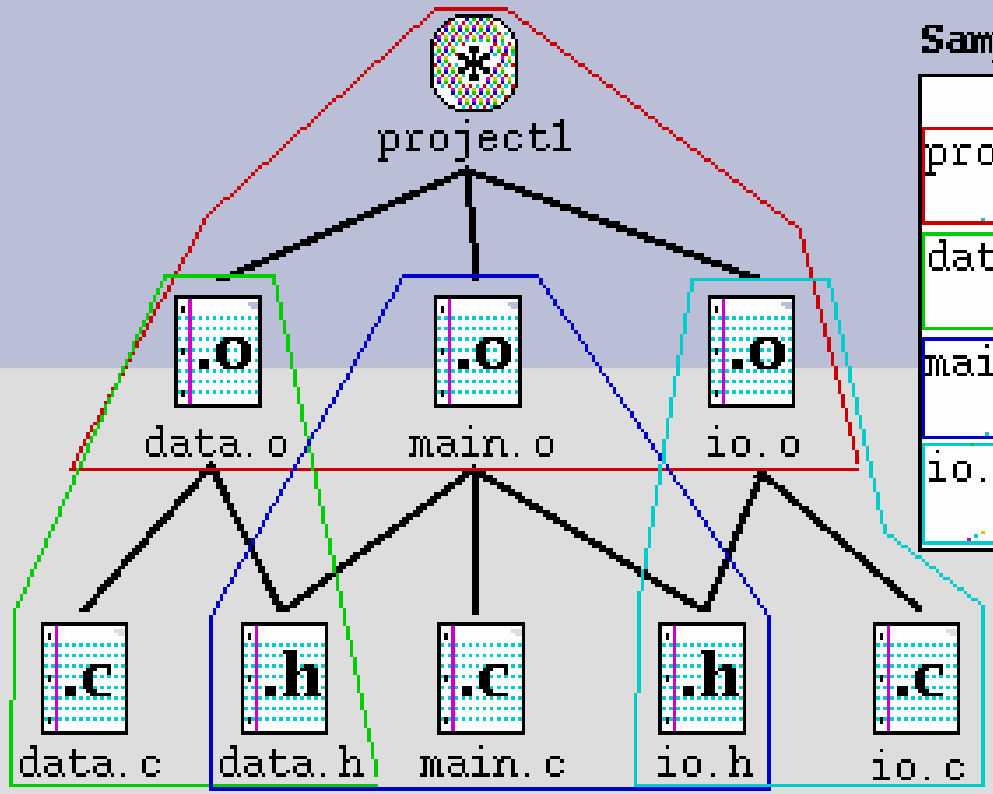


Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

For example, io.c was changed.

If you edit io.c, it becomes "newer" than io.o, meaning that make must run cc -c io.c to create a new io.o, then run cc data.o main.o io.o -o project1 for project1.



Sample Makefile

```
project1: data.o main.o io.o
    cc data.o main.o io.o -o project1
data.o: data.c data.h
    cc -c data.c
main.o: data.h io.h main.c
    cc -c main.c
io.o: io.h io.c
    cc -c io.c
```

Makefile uses the following format:

target : source file(s)
command (must be preceded by a tab)

A target given in the Makefile is a file which will be created or updated when any of its source files are modified. The command(s) given in the subsequent line(s) (which must be preceded by a tab character) are executed in order to create the target file.

Listing dependencies

```
project1: data.o main.o io.o
	cc data.o main.o io.o -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

Note that in the Makefile shown above, the .h files are listed, but there are no references in their corresponding commands.

Listing dependencies

```
project1: data.o main.o io.o
	cc data.o main.o io.o -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

This is because the .h files are referred within the corresponding .c files through the #include "file.h".
If you do not explicitly include these in your Makefile, your program will not be updated if you make a change to your header (.h) files.

Note: Comments can be placed in a Makefile by placing a pound sign (#) in front of it.

Using the Makefile with make

Once you have created your Makefile and your corresponding source files, you are ready to use make.
If you have named your Makefile either Makefile or makefile, make will recognize it.
If you do not wish to call your Makefile one of these names,

make -f mymakefile

The order in which dependencies are listed is important. If you simply type make and then return, make will attempt to create or update the first dependency listed.

You can also specify one of the other targets listed in the Makefile, and only that target (and its corresponding source files) would be made.

For example, if we typed make, the output of make would look as follows:

```
% make
        cc -c data.c
        cc -c main.c
        cc -c io.c
        cc data.o main.o io.o -o project1
%
```

When making its targets, make first checks the source files and attempts to create or update the source files.

That is why data.o, main.o and io.o were created before attempting to create the target: project1.

Shortcuts for make

The make program has many other features which have not been discussed in previous sections.

Most important of these features is the macro feature. Macros in make work similarly to macros used in C programming.

Make also has its own pre-defined rules which you can take advantage of to make your Makefile smaller.

Macros in make

The make program allows you to use macros, which are similar to variables, to store names of files. The format is as follows:

OBJECTS = data.o io.o main.o

Whenever you want to have make expand these macros out when it runs, type the following corresponding string $(OBJECTS).

Here is our sample Makefile again, using a macro.

```
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
	cc $(OBJECTS) -o project1
data.o: data.c data.h
	cc -c data.c
main.o: data.h io.h main.c
	cc -c main.c
io.o: io.h io.c
	cc -c io.c
```

You can also specify a macro's value when running make, as follows:
make 'OBJECTS=data.o newio.o main.o' project1

This overrides the value of OBJECTS in THE Makefile

Special macros

In addition to those macros which you can create yourself, there are a few macros which are used internally by the make program. Here are some of those, listed below:

CC
Contains the current C compiler. Defaults to cc.

CFLAGS
Special options which are added to the built-in C rule.

$@
Full name of the current target.

$?
A list of files for current dependency which are out-of-date.

Special macros(contd.)

$<

The source file of the current (single) dependency.

You can also manipulate the way these macros are evaluated, as follows, assuming that

OBJS = data.o io.o main.o,

using

$(OBJS:.o=.c)

within the Makefile substitutes .o at the end with .c, giving you the following result:

data.c io.c main.c

Predefined rules

By itself, make knows already that in order to create a .o file, it must use cc -c on the corresponding .c file.

These rules are built into make, and you can take advantage of this to shorten your Makefile.

If you just indicate just the .h files in the dependency line of the Makefile that the current target is dependent on, make will know that the corresponding .c file is already required. You don't even need to include the command for the compiler.

This reduces our Makefile further, as shown:

```
OBJECTS = data.o main.o io.o

project1: $(OBJECTS)
        cc $(OBJECTS) -o project1

data.o: data.h

main.o: data.h io.h

io.o: io.h
```

you may wish to add a CFLAGS macro at the top of your Makefile to enable the compiler to use C compiler options. The macro looks like this:

```
CFLAGS=-Aa -D_HPUX_SOURCE
```

Miscellaneous shortcuts

Although the examples we have shown do not explicitly say so, you can put more than one file in the target section of the dependency rules.
If a file appears as a target more than once in a dependency, all of its source files are included as sources for that target.

Here is our sample Makefile again:

```
CFLAGS = -Aa -D_HPUX_SOURCE
OBJECTS = data.o main.o io.o
project1: $(OBJECTS)
        cc $(OBJECTS) -o project1
data.o main.o: data.h
io.o main.o: io.h
```

Advanced features

Make has some advanced features which we will discuss very briefly.

This includes specialized dependency rules, as well as writing Makefiles to work with your own suffixes, and writing new "built-in" rules.

Special dependencies

Usually, make uses the same command to create or update a target, regardless of which file changes.
Some other files, such as libraries allow users to replace a portion of its code.
For this kind of different behavior, make allows a special form of the dependency, where the action specified can differ, depending on which file has changed. Here is an example for this rule:

```
target :: source1
    command1
target :: source2
    command2
```

As we have described, if source1 changes, target is created or updated using command1; command2 is used if source2 is modified instead.

Custom suffixes and rules

Make uses a special target, named .SUFFIXES to allow you to define your own suffixes.
For example, the dependency line:

```
.SUFFIXES: .foo .bar
```

tells make that you will be using these special suffixes to make your own rules.

Custom suffixes and rules

Similar to how make already knows how to make a .o file from a .c file, you can define rules in the following manner:

```
.foo.bar:
	tr '[A-Z][a-z]' '[N-Z][A-M][n-z][a-m]' < $< > $@
.c.o:
	$(CC) $(CFLAGS) -c $<
```

The first rule allows you to create a .bar file from a .foo file. (Don't worry about what it does, it basically scrambles the file.)

The second rule is the default rule used by make to create a .o file from a .c file.