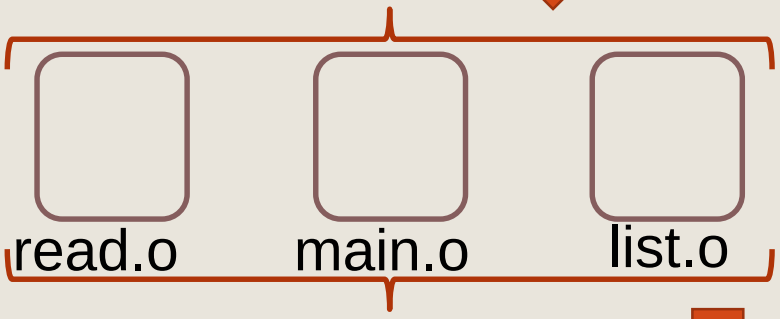
Multiple file project management & Makefile

Compilation & linkage

Compilation:

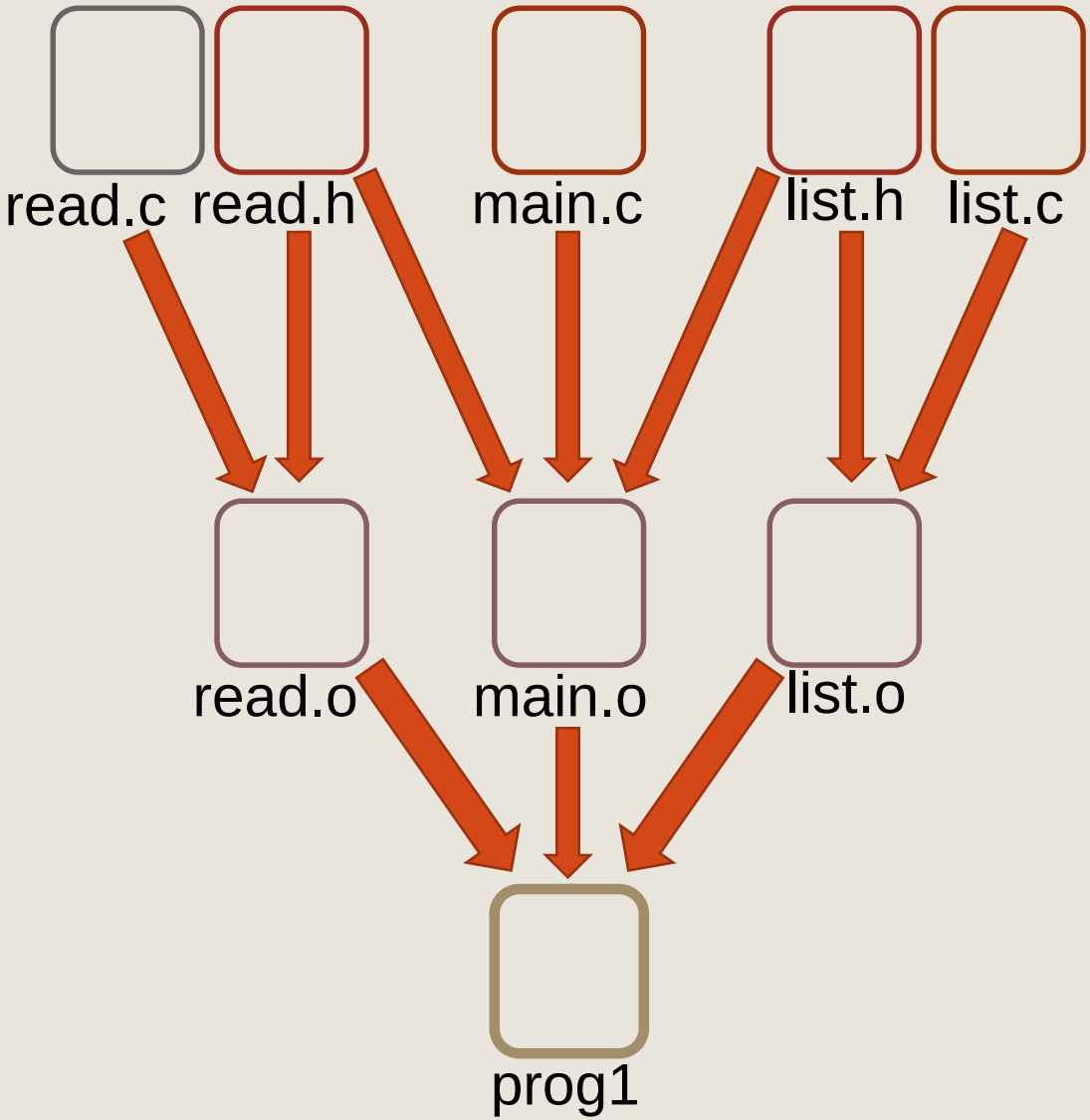
gcc … -c read.c main.c list.



Linkage:

gcc ... read.o main.o list.o -o prog1

Compilation & linkage

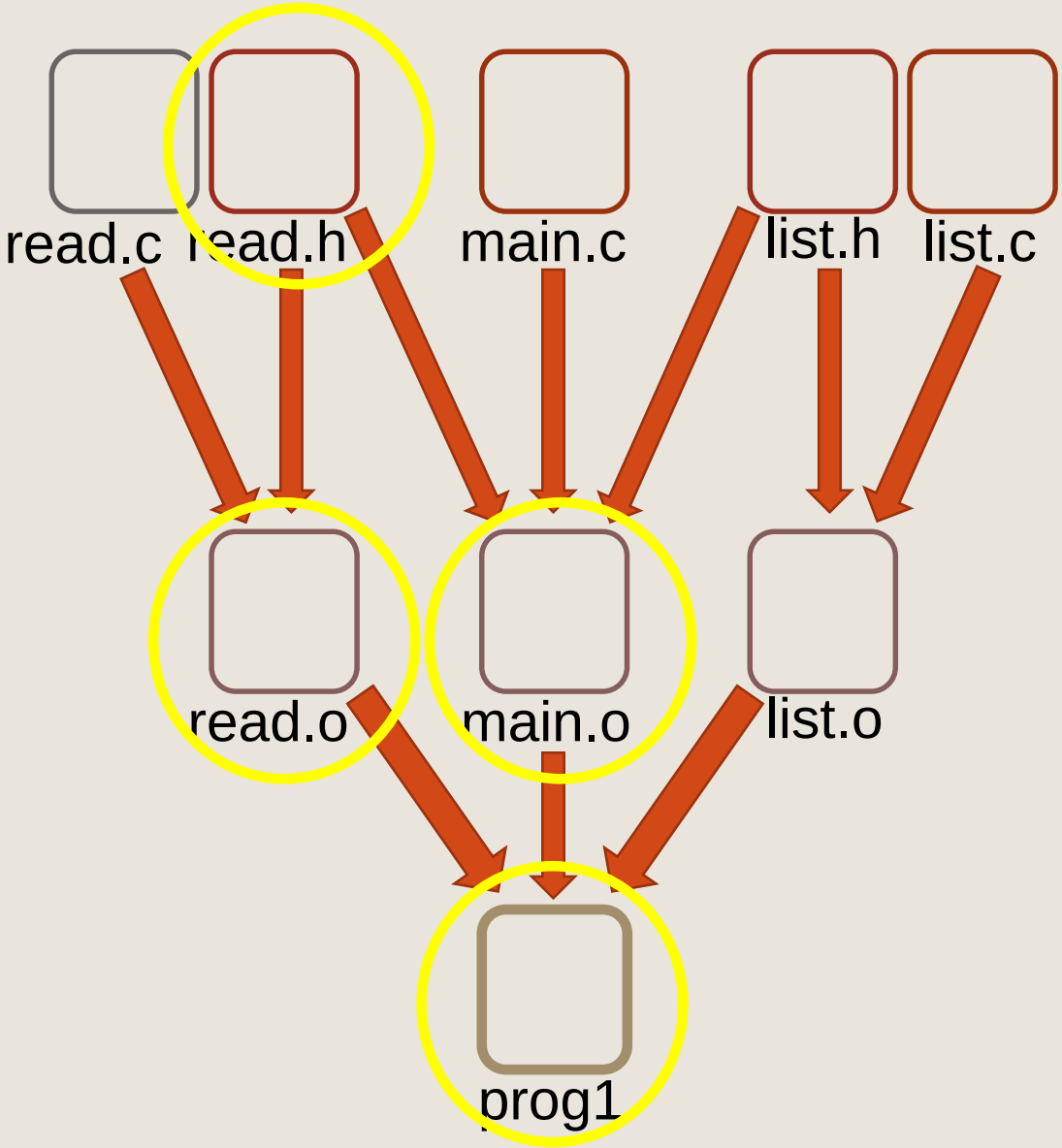


Compilation & linkage

If only one file is modified, will we have to recompile all over again?

- No!
- **Dependencies tree**

Example



(gnu) make

(gnu) make

make is a program who's main aim is to update other programs in a "smart" way.

"smart" =

- Build only out-of-date files (use timestamps).
- Use the dependency graph for this.

You tell make what to do by writing a "makefile"

makefile names

make looks automatically for : `makefile, Makefile`

Override by using –f :

```
make -f MyMakefile
```

What makefile contains

- Explicit rules
- Implicit rules
- Variables/MACROS definitions
- Comments

Makefile

Aim: Build only out-of-date files (use timestamps)

Makefile contains:

- List of dependencies (no cycles)
- “Recovery” scenario when any file is modified

```
main.o: main.c list.h read.h
	gcc -c main.c
```

Beware of the essential **tab**!

In words: if any of the files {main.c, list.h, read.h} was modified after main.o, the command “gcc -c main.c” will be performed

Explicit rules

Rule Syntax:

```
targets :  prerequisites
 <tab> shell command
 <tab> ...
```

- Targets:
  Usually files
- Prerequisites:
  Other files (or targets) that the targets depend on
- Command:
  What to execute (shell command)
  <tab> tells make it’s a command

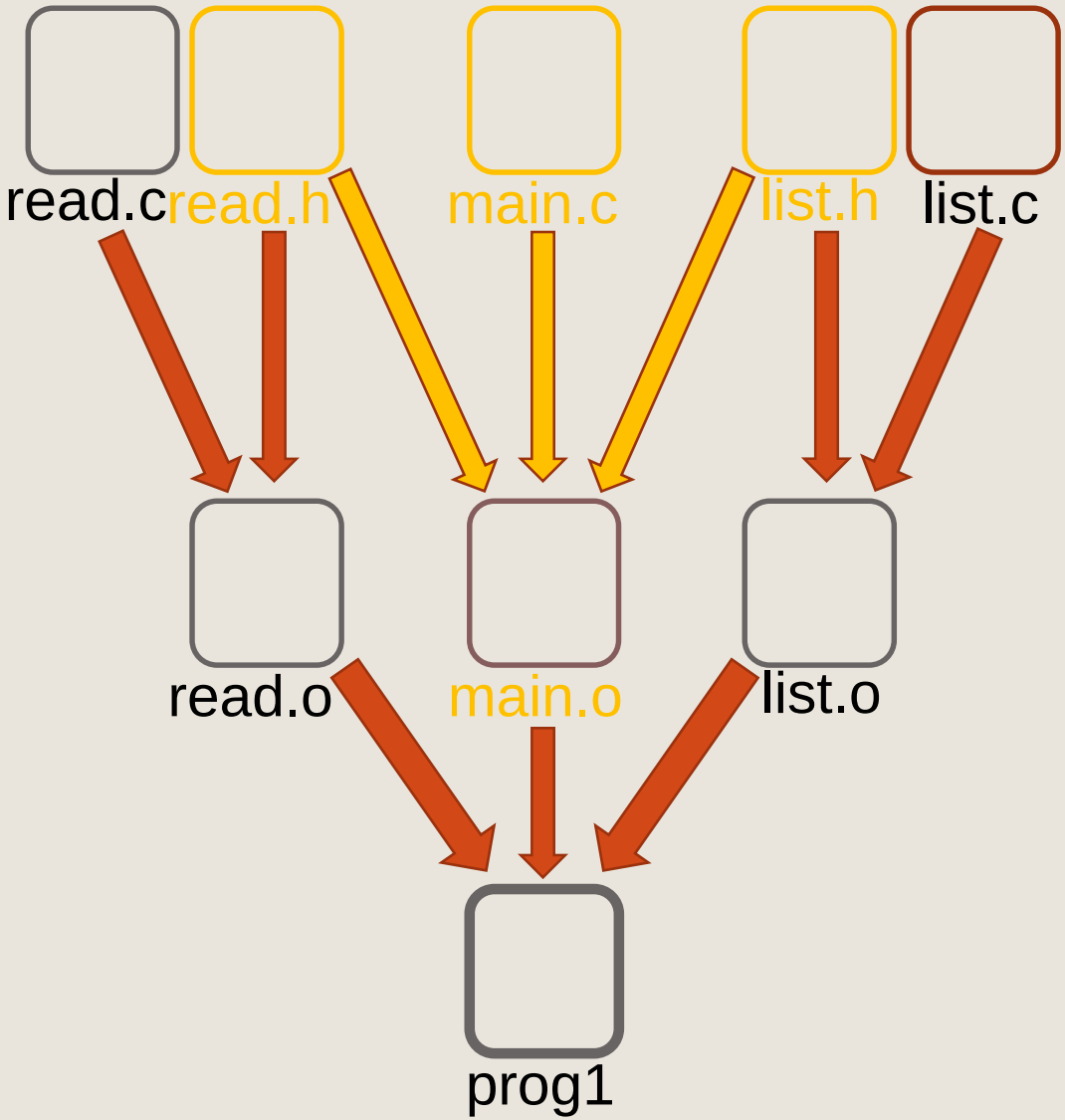
How make works?

Given a target:

1. Find the rule for it
2. Check if the rule prerequisites are up to date
   - By “recursive” application on each of them
3. If one of the prerequisites is newer than target run the command in rule body.

- Use the flag <-n> to print the commands being performed.

Dependencies Tree



Example

```
prog1: main.o read.o list.o
  gcc main.o read.o list.o –o prog1

main.o: main.c read.h list.h
  gcc -c main.c

read.o: read.c read.h
  gcc -c read.c

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

Example

```
prog1: main.o read.o list.o
  gcc main.o read.o list.o –o prog1

main.o: main.c read.h list.h
  gcc -c main.c

read.o: read.c read.h
  gcc -c read.c

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

Running make: **make prog1**

If **read.h** modified:

```
Check prog1
   Check: main.o
          Do: gcc -c main.c
   Check read.o
       Do: gcc -c read.c
   Check: list.o
        Do: <nothing>
   Do: gcc main.o read.o
            list.o –o prog1
```

Comments

```
# comments are easy
```

Variables/MACROS

Make maintains variables

- Case sensitive
- Traditionally – use capital letters
- Many automatically defined with default values.

```
FOO = 1
CFLAGS = -Wall
TARGETS = list.o main.o read.o
```

Rules can use variables

```
main.o: main.c read.h list.h
  gcc $(CFLAGS) –c main.c
```

Automatic Variables

Set automatically by make, depend on current rule

$@ - target

$^ - list of all the prerequisites

- including the directories they were found

$< - first prerequisite in the prerequisite list.

- $< is often used when you want just the .c file in the command, while the prerequisites list contains many .h too

$? - all the prerequisites that are newer than the target

Many Others …….

Using Wildcards

Automatic Wildcard (*,?) expansion in:

- Targets
- Prerequisites
- Commands

```
clean:
 rm -f *.o  # good

objects = *.o  # no good
```

Implicit Rules

When no explicit rule defined, an implicit rule will be used.

- not always sufficient (e.g. doesn’t check .h files update)

Implicit Rules

We saw “explicit rules” so far, e.g:

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

Implicit rules (many kinds):

- Built-in - create “.o” files from “.c” files.

```
foo: foo.o bar.o
        gcc –o foo foo.o bar.o
```

No need to tell make to create o from c

- If we would like to write this explicitly (not needed!)

```
%.o : %.c
    gcc –c $< # -o $@
```

$@ - file for which the match was made (e.g. list.o)

$< - the matched dependency (e.g. list.c)

Implicit Rules

We can write:

```
prog1: main.o read.o list.o
     gcc main.o read.o list.o –o prog1

main.o: read.h list.h

read.o: read.h

list.o: list.h
```

The dependencies of main.o on main.c are specified in the rule, and also the command to build them.

Implicit Rules

One more example for implicit rule:

```
%.class: %.java
        javac $<
```

- Result: For every “.java” file that was modified, a new “.class” file will be created.

Automatic dependencies

gcc know to resolve dependencies:

```
$ gcc –MM main.c
main.o: main.c read.h list.h
```

Automatic add dependencies to the makefile:

```
gcc -MM *.c >> makefile
```

Also read on: makedepend

Makefiles: all, clean, .PHONY

```
#Makes all progs
all: prog1, prog2
        command
...
# Not really file names
.PHONY: clean all
# Removing the executables and object files
clean:
        rm prog1 prog2 *.o
```

.PHONY – not really a file name.

Automatic makefiles

Many modern IDEs there is no need to write makefiles. They are created for you (eclipse, Visual studio)

It is good to understand what's going on when compiling. So, **write your own makefiles**

Example: List Makefile

Program Design

Interfaces

A definition of a set of functions that provide a coherent module (or library)

- Data structure (e.g., list, binary tree)
- User interface (e.g., drawing graphics)
- Communication (e.g., device driver)

Reminder: List

Interface - modularity

Hide the details of implementing the module from its usage

- Specification – “what”
- Implementation – “how”

Interface – information hiding

Hide “private” information from outside

- The “outside” program should not be able to use internal variables of the module
- Crucial for modularity

Resource management

- Define who controls allocation of memory (and other resources)

Interface Principles

Hide implementation details

1. Hide data structures
2. Don’t provide access to data structures that might be changed in alternative implementation
3. A “visible” detail cannot be later changed without changing code using the interface!

Interface Principles

Use small set of “primitive” actions

1. Provide to maximize functionality with minimal set of operations
2. Do not provide unneeded functions “just because you can”
3. How much functionality? Two approaches: Minimal (For few users, don't waste your time / Maximal (when many users will use it e.g. OS)

Interface Principles

Don’t reach behind the back

1. Do not use global variables unless you must.
2. Don't have unexpected side effects!
3. Use comments if you assume specific order of operations by the user (and force it).

Interface Principles

Consistent Mechanisms

1. Do similar things in a similar way

   - strcpy(dest, source)
   - memcpy(dest, source)

Interface Principles

Resource Management

1. Free resource at the same level it was allocated – the one how allocates the resource is responsible to free it.
2. Assumptions about resources

Libraries

Libraries

A library is a collection of code (functions, global variables, etc.) written (sometimes also compiled) by someone else, that you may want to use

Examples:

- C’s standard libraries
- Math library
- Graphic libraries

Compiled Libraries

Compiled libraries may be composed of many different object files

Compiled Libraries

Static libraries:

- linked with your executable at compilation time
- standard unix suffix: .a (windows: .lib)

Shared libraries:

- loaded by the executable at run-time
- standard unix suffix: .so (windows: .dll)

Static libraries – first pass

Using the static library libdata.a:

```
gcc object1.o -ldata -o prog
```

Creating the library data.a (2 commands):

```
ar rcu libdata.a data.o stack.o list.o
ranlib libdata.a
```

ar is like tar – archive of object files

ranlib builds a symbol table for the library

- to be used by the linker

static vs. shared

Static libraries pros:

1. Independent of the presence/location of the libraries
2. Less linking overhead on run-time

Shared libraries pros:

1. Smaller executables
2. Multiple processes share the code
3. No need to re-link executable when libraries are changed
4. The same executable can run with different libraries
5. Dynamic Library Loading (dll) possible

static vs. shared

BEWARE: dll hell: exectuables running with incorrect versions of the library,
to unexpected results…

The same executable can run with different libraries

Libraries in makefile

```
libdata.a:  ${LIBOBJECTS}
    ar rcu libdata.a ${LIBOBJECTS}
    ranlib libdata.a


OBJECTS = foo.o bar.o
CC = gcc
prog: ${OBJECTS} libdata.a
    ${CC} ${OBJECTS} –ldata –o prog
```

Static libraries – second pass

```
#include <util.h> // Library header
int main ()
{
   foo(); // foo is a function of the
          // 'util' library
   ...
}
```

Static libraries – second pass

Compilation:

```
gcc -Wall -c –I /usr/include/myCode/
    main.c –o main.o
```

Linking:

```
gcc main.o -L /usr/lib/myLib/
   -lutils -o app
```

The linking process: Objects vs. static libraries

objects:

- The whole object file linked to the executable, even when its functions not used.
- Two function implementations – will cause error.

Libraries:

- Just symbols (functions) which not found in the obj files are linked.
- Two function implementations – first in the obj file, and second in library – The first will be used.

Dynamic Libraries

Library creation:

Compilation:

```
 gcc -Wall –fPIC -c utils.c
```

Linking:

```
 gcc –shared utils.o –o libutils.so
```

Usage:

```
Linking to:
 gcc –Wall –c –I/usr/lib/include
     -L/usr/lib/bin –lutils –o prog
```

*PIC – position independent code.

** On windows you need __declspec(dllimport) …

Dynamic Libraries

- Why do we link at compile time to dynamically linked library?
- Not real linking, just to check that linking is possible. Actual linking is done in run time.
  => Need to know how to find in runtime. Should be in the dynamic library search path. (e.g. c:\windows\system)

- True dynamic loading (i.e. browser plugins) service is provided by OS (dlopen on linux, loadlibrary on windows)