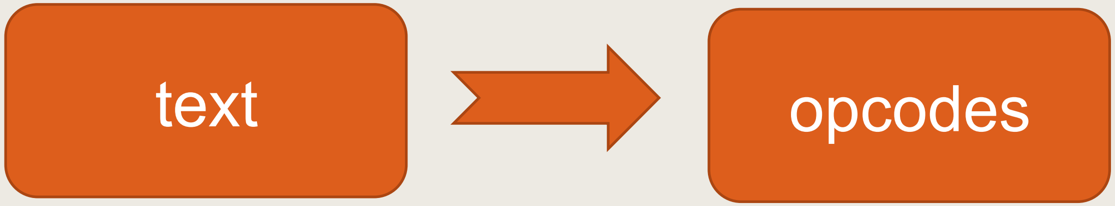
Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 6-7
6-8.11.2018

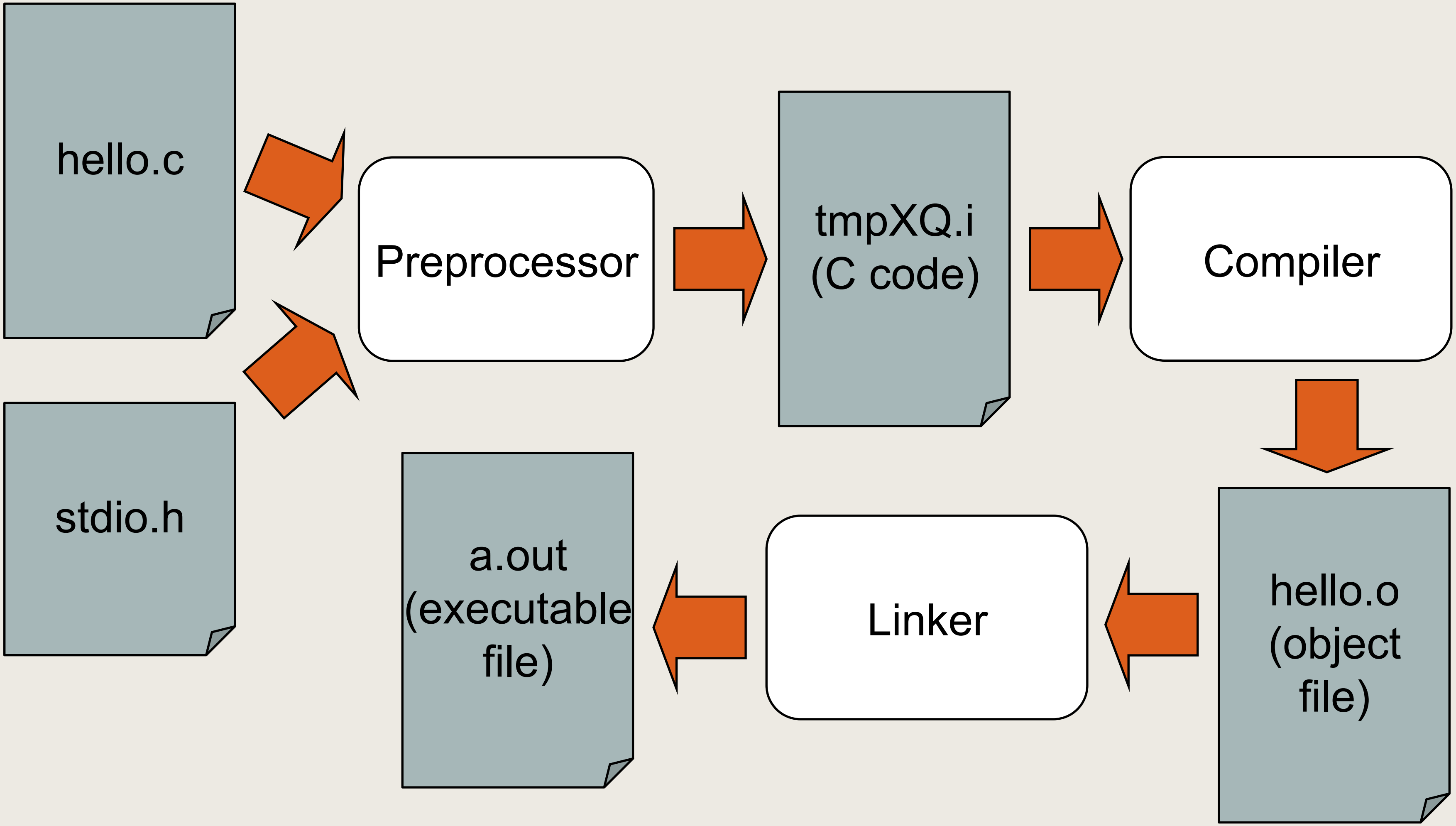
Compilation and Linkage



text

opcodes

Code to executable in C



* gcc is called “compiler”, though it also runs the preprocessing and (optionally) linking

Code to executable in C

- The **C Preprocessor** is not a part of the compiler
  - separate step in the compilation process
  - preprocessor is just a text substitution tool

Preprocessor

- **Compilation** is the processing of source code files (.c, .cc, or .cpp) and the creation of an 'object' files (*.o)
  - You can’t run object files
  - The compiler produces the machine language instructions

Compiler

- **Linking** creates a single executable file from multiple object files
  - linker will complain about undefined functions. If the compiler could not find the definition for a particular function, it would just assume that the function was defined in another file.
  - The linker may look at multiple files and try to find references for the functions that weren't mentioned

Linker

The Preprocessor

Preprocessor

A single-pass program that:

1. Includes header files
2. Expands macros
3. Controls conditional compilation
4. Removes comments

Outputs –
a code ready for the compiler to work on

Preprocessor

We can test what the preprocessor does:

```
> gcc -E hello.c
```

will print the C code after running the preprocessing stage

Common Pre-processing Directives

preprocessor commands begin with a hash symbol (#)

- `#include`
- `#define`
- `#if ... #else ... #endif`

#include directive

```
#include "foo.h"
```

Includes the file ‘foo.h’, starts searching from the **same directory as the current file** (the one that contains the `#include` directive)

```
#include <stdio.h>
```

Includes the file ‘stdio.h’, starts searching from the **standard library directory** (part of gcc installation)

#include directive

```
#include “file”
```

=

Copy & paste the content of “file” in this location and continue with pre-processing

Header files

Header file contains

1. Definitions of data types (`typedef`, `structs`)
2. Declarations of functions & constants that are shared by multiple modules

`#include` allows several modules to share the same set of definitions/declarations

Modules & Header files

Square.h

```
// declaration
int area (int x1, int y1, int x2, int y2);
int length (int x1, int y1, int x2, int y2);
...
```

Square.c

```
#include “Square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y2){
   return length(x1,y1,x2,y1) *
          length(x1,y2,x1,y2);
}
...
```

MyProg.c

```
#include “Square.h"
int main()
{
  // usage
  area (2,3,5,6);
}
```

#define directive

```
#define FOO 1
```

```
int x = FOO;
```

is equivalent (after the preprocessing) to:

```
int x = 1;
```

#define with arguments - MACRO

```
#define SQUARE(x) x*x
b = SQUARE (a);
```

is the same as

```
b = a*a;
```

#define - cautions

```c
#define SQUARE(x) x*x
#define PLUS(x) x+x
b = SQUARE(a+1);
c = PLUS(a)*5;
```

Is it what we intended?

#define - cautions

```
#define SQUARE(x) x*x
#define PLUS(x) x+x
b = SQUARE(a+1);
c = PLUS(a)*5;
```

We actually get the following:

```
b = a+1*a+1; // b = 2*a + 1
c = a+a*5;   // c = 6*a
```

#define - cautions

```
#define SQUARE(x) x*x
#define PLUS(x) x+x
b = SQUARE(a+1);
c = PLUS(a)*5;
```

Solution:

```
#define SQUARE(x) ((x)*(x))
#define PLUS(x) ((x)+(x))
```

#define

Multi-line:

All preprocessor directive effect one **line** (not c statement).

To insert a line-break, use “\”:

BAD:

```
#define x (5 +
           5)
```

GOOD:

```
#define x (5 + \
           5)
// x == 10 !
```

What are the disadvantages of macros?

- Macros can’t be debugged, many debuggers can’t see what the macro translates to
- Macro expansions can have side effects

Alternative to macros

- **Constants**

```
enum { FOO = 1 }; // will be discussed later
```

or

```
const int FOO = 1;
```

- **Functions** – inline functions (C99, C++, will discuss this later on)

#if directive: conditional compilation

```
#define DEBUG

…

#if defined(DEBUG)
  // compiles only when DEBUG exists (defined)
  printf("X = %d\n", X);
#endif
```

Debugging - assert

Example of using **conditional compilation**

assert.h

```
#include <assert.h>
// Sqrt(x): compute square root of x
// Assumption: x is non-negative
double sqrt(double x )
{
   assert( x >= 0 );  // aborts if x < 0
...
```

If the program violates the condition, then the program will abort and print:

```
assertion "x >= 0" failed: file "Sqrt.c",
line 7 <exception>
```

assert.h

The assertion allows to catch the event in debug mode, during run-time (**not** compilation time)!

assert.h

- Important coding practice
- Declare implicit assumptions
- Sanity checks in code
- Check for violations during debugging/testing

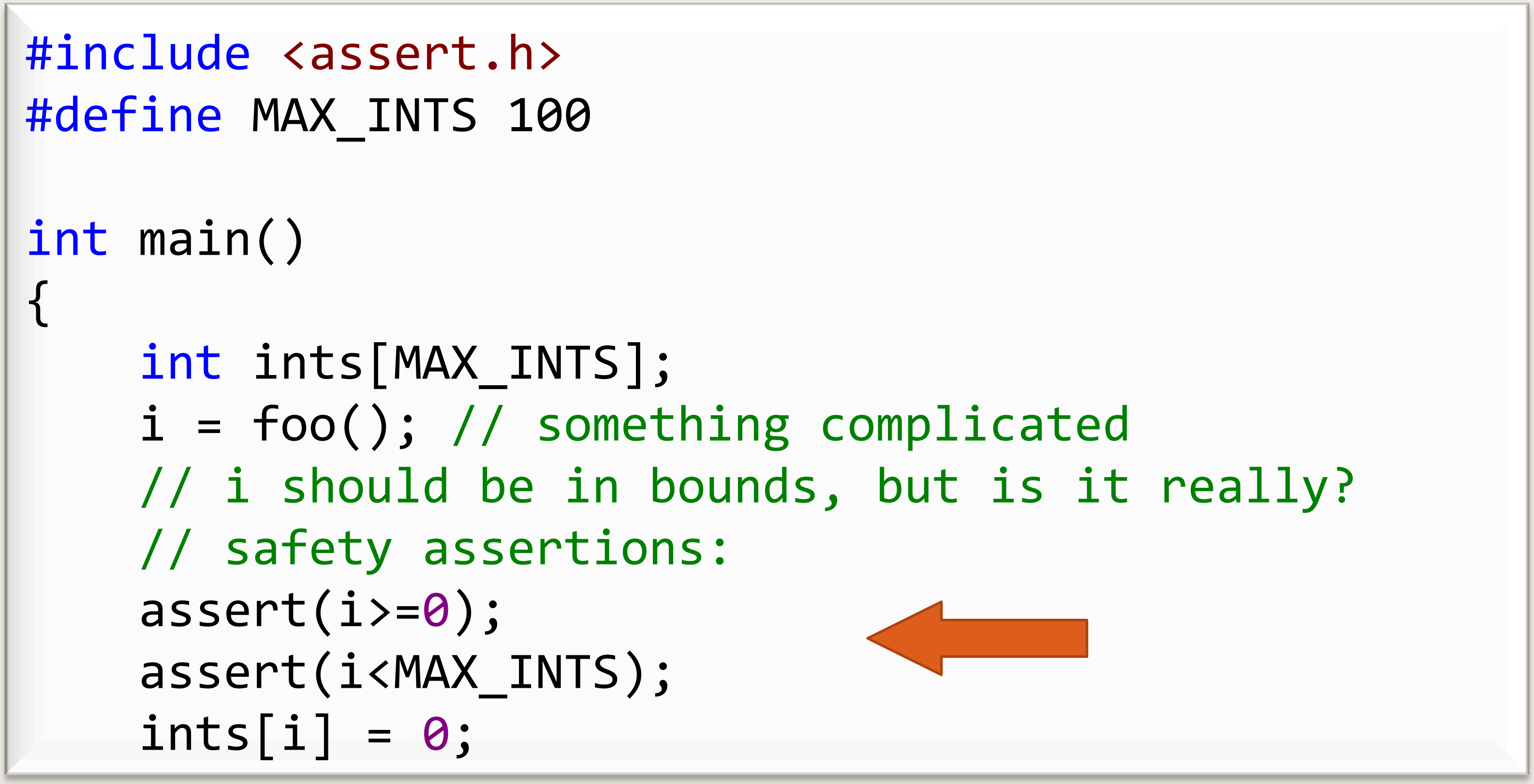
The following examples include more preprocessing directives (#, ##) – read at home about this syntax

assert.h

```
// procedure that prints error message // to disable
the printing define the macro NDEBUG
// before the <assert.h> inclusion
void __assert(char* file, int line, char* test);


#ifdef NDEBUG
   #define assert(e)     ((void)0)
#else
   #define assert(e)     \
      ((e) ? ((void)0) : \
      __assert(__FILE__, __LINE__, #e))
#endif
```

Debug/Test mode vs Release mode



```
#include <assert.h>
#define MAX_INTS 100

int main()
{
    int ints[MAX_INTS];
    i = foo(); // something complicated
    // i should be in bounds, but is it really?
    // safety assertions:
    assert(i>=0);
    assert(i<MAX_INTS);
    ints[i] = 0;
```

Debug/Test mode vs Release mode

```
#define NDEBUG
#include <assert.h>
#define MAX_INTS 100

int main()
{
    int ints[MAX_INTS];
    i = foo(); // something complicated
    // should be in bounds, but is it really?
    // safety assertions
    ~~assert(i>=0);~~
    ~~assert(i<MAX_INTS);~~
    ints[i] = 0;
    ...
```

Defining NDEBUG using the compiler

```
>> gcc my_program.c -DNDEBUG -o my_exe
```

This is equivalent for adding at the beginning of the file, the definition:

```
#define NDEBUG
```

Preprocessor – summary

- **Text** processing program
- Does not know C language rules
- Operates before compilation, output passed to compiler
- Can do “copy and paste”, or, “cut”

Preprocessor – summary

`#include`

- pastes the included file to current file (.h by convention)
- usually contains forward declarations and type definitions

`#define`

- copy-pastes the macro body where macro name appears
- used for constants, or simple "functions"

`#if`

- if condition is not fulfilled, “cut” the code
- conditional compilation (e.g. debugging code)

Compilation

Modules & Header files

Square.h

```
// declaration
int area (int x1, int y1, int x2, int y2);
...
```

MyProg.c

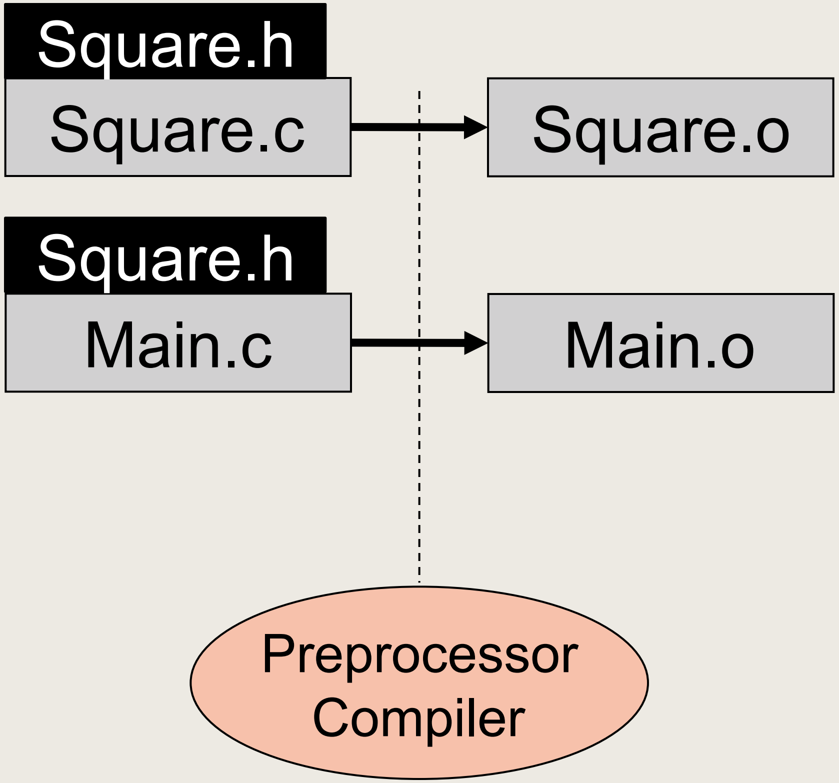
```
#include "square.h"
int main()
{
  // usage
  area (2,3,5,6);
}
```

Square.c

```
#include “Square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y2)
{
...
}
```

Compiling

- Creates an object file for each code file (.c -> .o)
- Each .o file contains opcode of the C code of its *translation unit* (functions, structs, variables etc..)
- Unresolved references still remain

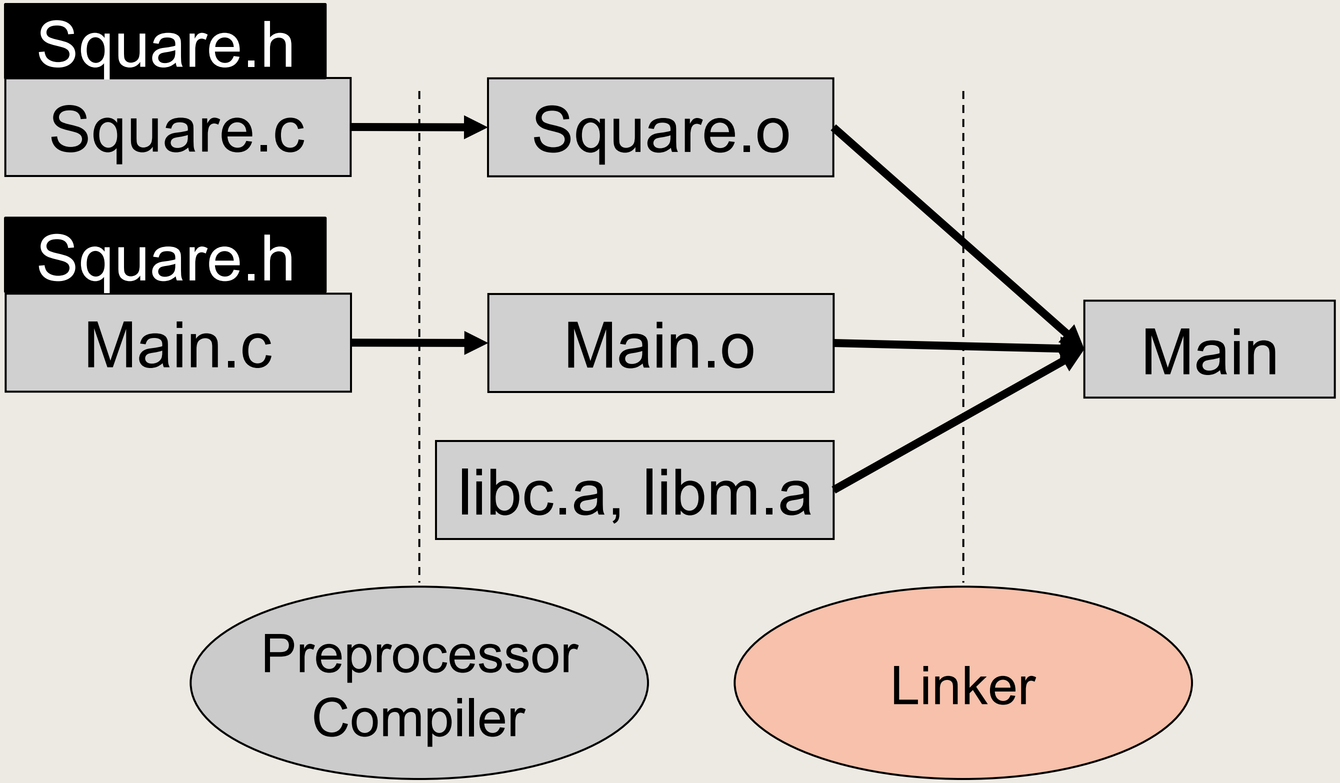


Linking

Combines several object files into an executable file

No unresolved references should remain

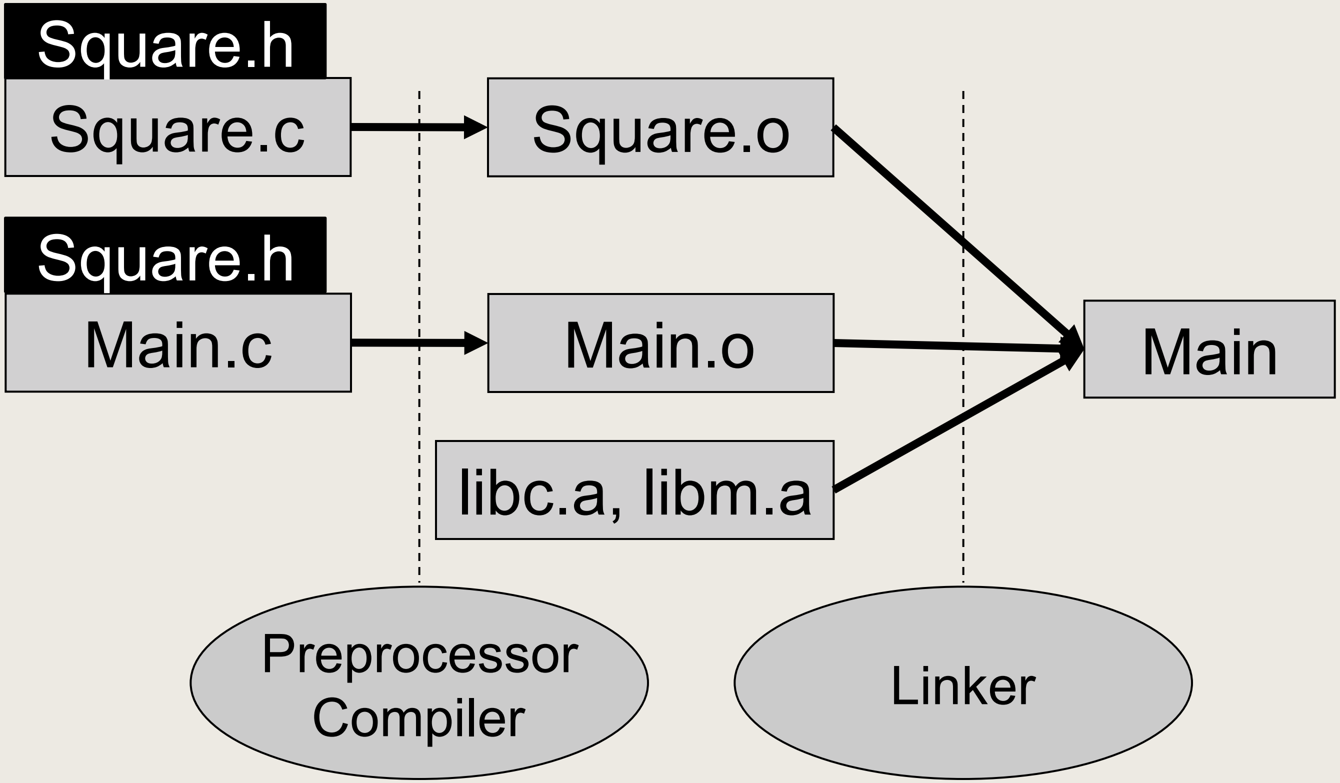
- Link function calls to function definition code
- Assign symbols to memory addresses



The whole process

Do not run the linker

```
$ gcc -c -Wall Square.c -o Square.o
$ gcc -c -Wall Main.c -o Main.o
$ gcc Square.o Main.o libc.a libm.a -o Main
```



Link errors

The following errors appear only at link time

1. Missing implementation

```
> gcc –Wall –o Main Main.c
Main.o(.text+0x2c):Main.c: undefined
reference to `foo'
```

2. Duplicate implementation (in separate modules)

```
> gcc –Wall -o Main Main.o foo.o
foo.o(.text+0x0):foo.c: multiple definition of
`foo'
Main.o(.text+0x38):Main.c: first defined here
```

Header safety

Structs – poor oop

complex.h

```c
struct Complex
{
    double _real, _imag;
};
struct Complex addComplex(struct Complex, struct Complex);
```

complex.c

```c
#include "complex.h"
// implementation
struct Complex addComplex(struct Complex a, struct Complex b)
{ ...
```

MyProg.c

```c
#include "complex.h"
int main()
{
    struct Complex c;
    ...
```

Header safety

Complex.h:

```
struct Complex
{ ... };
```

MyStuff.h:

```
#include "Complex.h"
```

Main.c:

```
#include "MyStuff.h"
#include "Complex.h"
```

Error:
Complex.h:1: redefinition of `struct Complex'

Header safety

Complex.h (revised):

```
#ifndef COMPLEX_H
#define COMPLEX_H
struct Complex
{
...
#endif
```

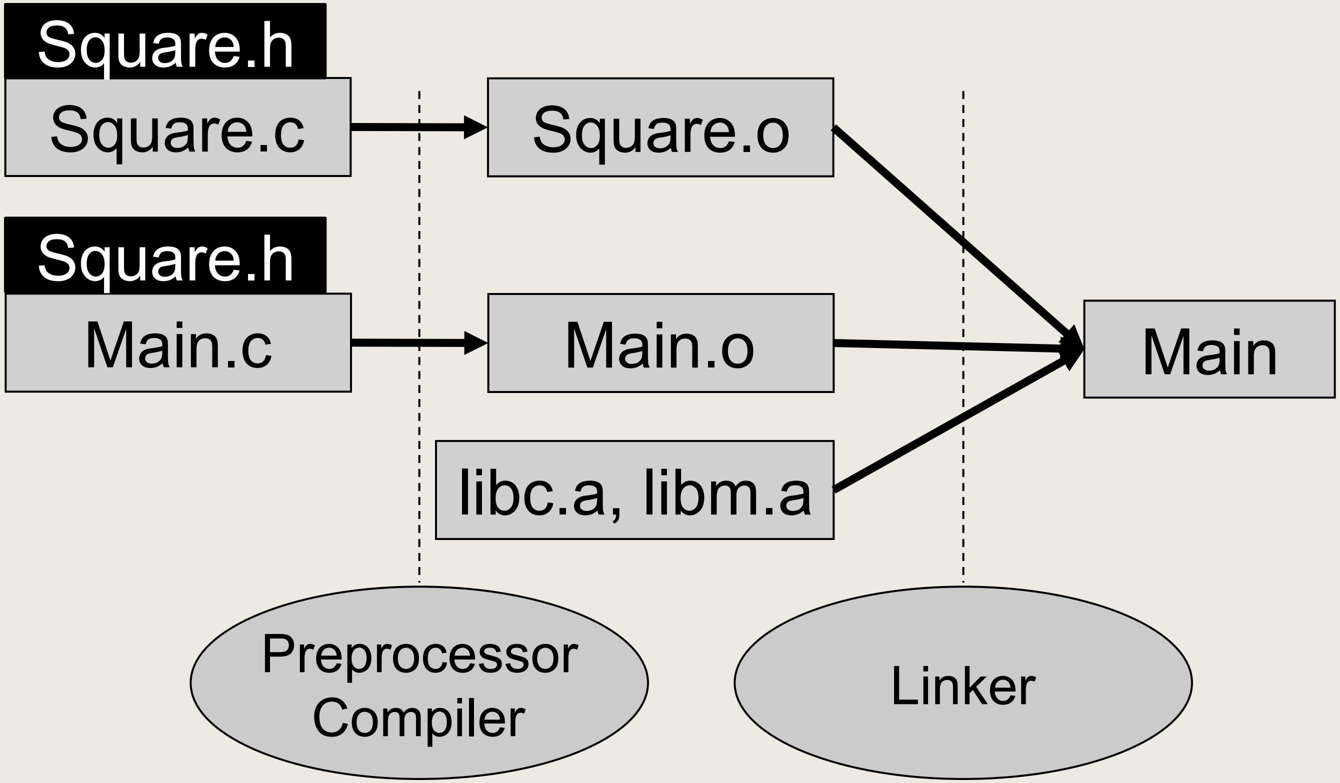
Main.c:

```
#include "MyStuff.h"
#include "Complex.h" // no error this time – why?
```

Multiple file project management

The whole process

```
$ gcc -c -Wall Square.c -o Square.o
$ gcc -c -Wall Main.c -o Main.o
$ gcc Square.o Main.o libc.a libm.a -o Main
```



Make

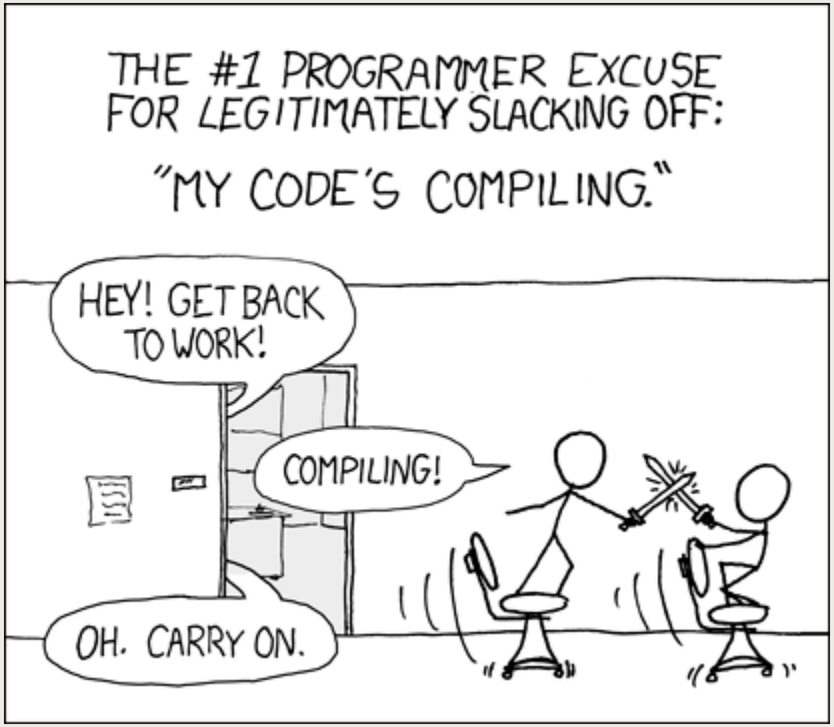
What is it?

- **Automatic** tool for projects management (not just C/C++)

What is it good for?

- Faster compilation/linkage => more productivity!
- Less boring work for the programmer => Less errors!

- man/google/**gnu** make



Make and Makefiles

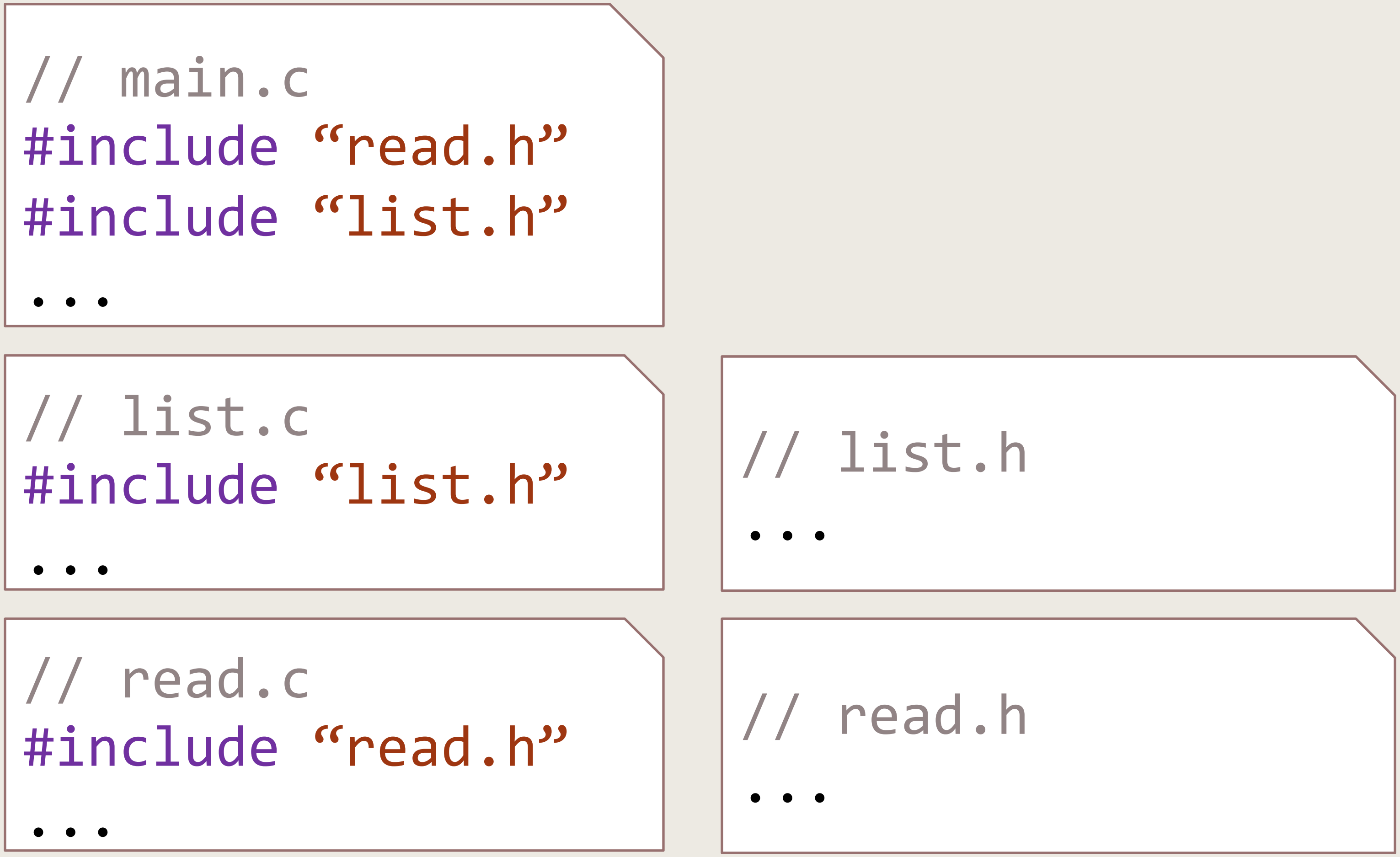
Make is a program who’s main aim is to update other programs in a “smart” way

“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*

Compilation & linkage



```
// main.c
#include “read.h”
#include “list.h”
...
```

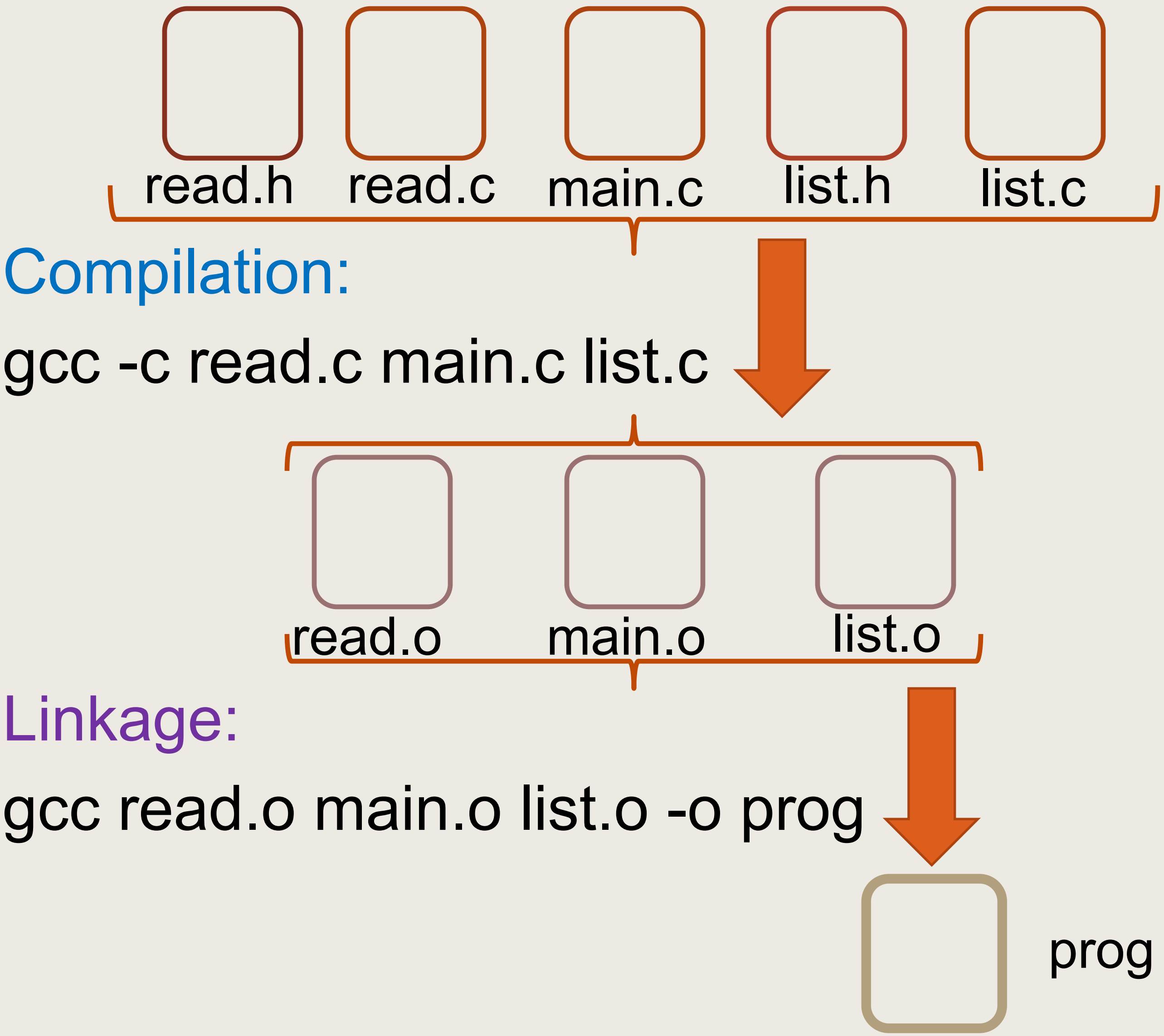
```
// list.c
#include “list.h”
...
```

```
// list.h
...
```

```
// read.c
#include “read.h”
...
```

```
// read.h
...
```

Compilation & linkage



read.h read.c main.c list.h list.c

Compilation:

```
gcc -c read.c main.c list.c
```

read.o main.o list.o

Linkage:

```
gcc read.o main.o list.o -o prog
```

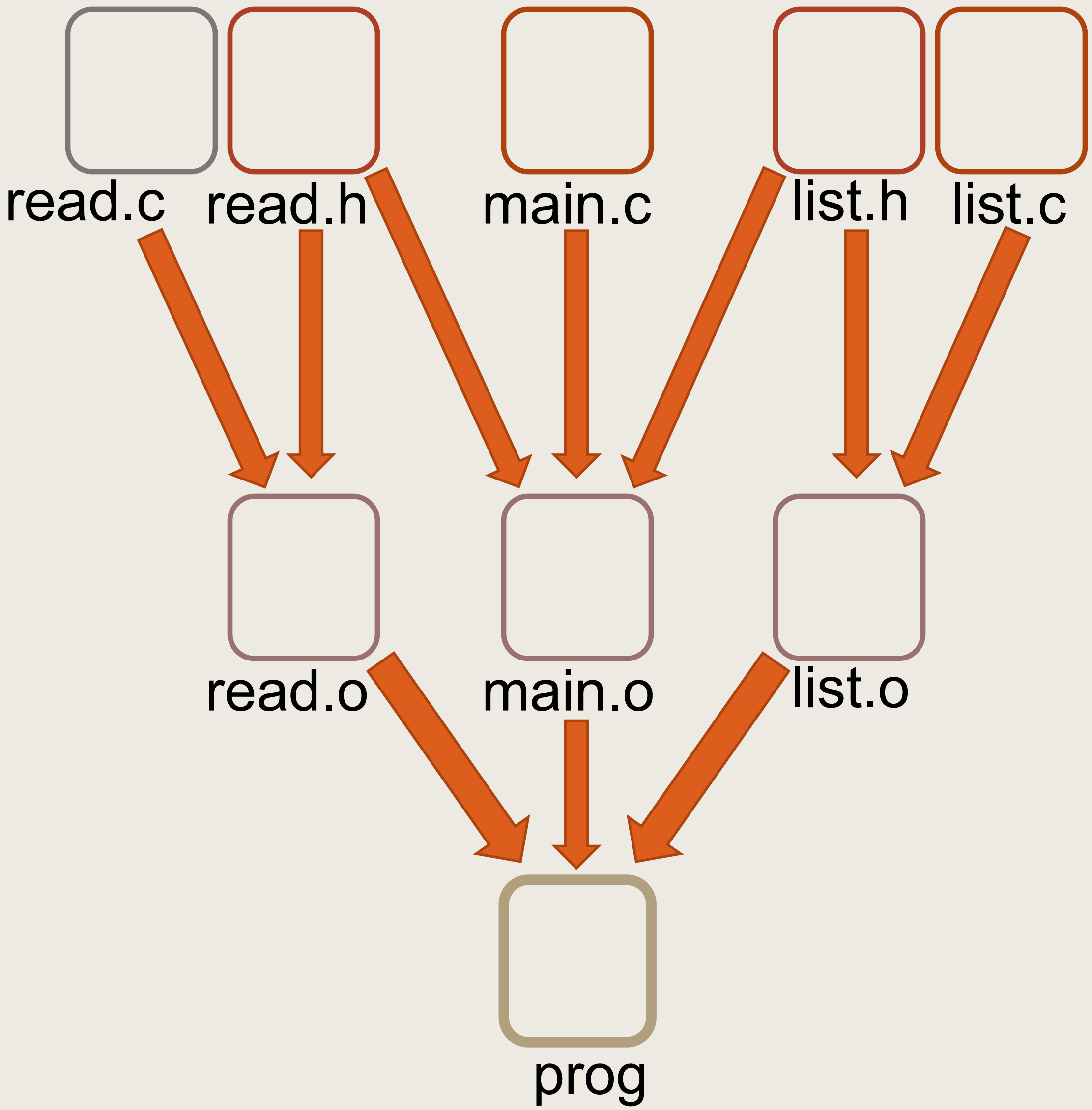
prog

Compilation & linkage

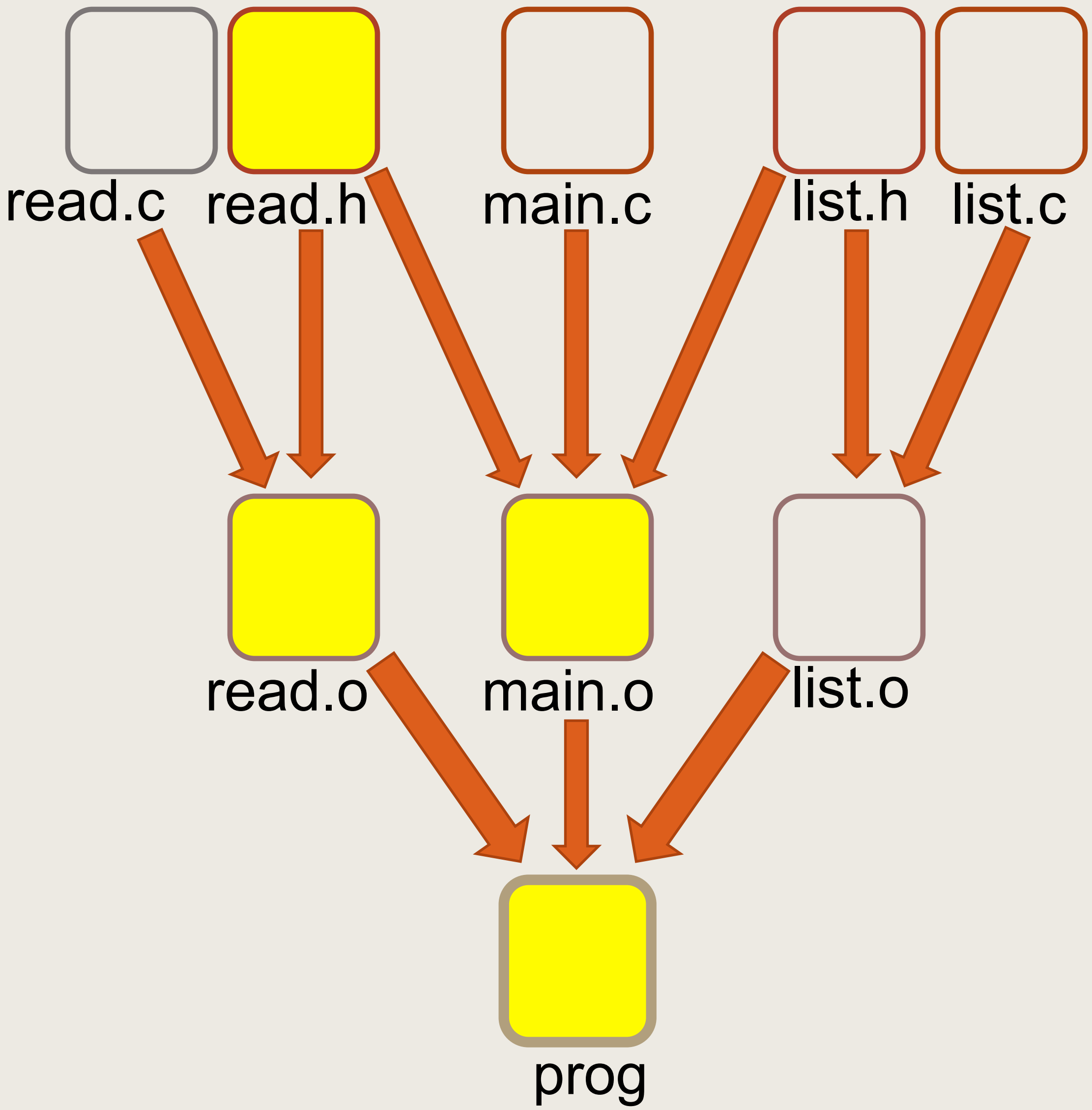
If only one file is modified, will we have to recompile all over again?

- No!
- **Dependencies tree**

Dependencies Tree



read.h change implication



Makefile

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

Format:

```
#comment
target: dependencies
[tab] system command
[tab] system command
...
```

Beware of the essential **tab**!

Running make examples:

```
> make prog
> make
> make -f myMakefile
```

Modules & Header files

Square.h

```c
// declaration
int area (int x1, int y1, int x2, int y2);
int length (int x1, int y1, int x2, int y2);
...
```

MyProg.c

```c
#include “Square.h"
int main()
{
  // usage
  area (2,3,5,6);
}
```

Square.c

```c
#include “Square.h"
#include <math.h>
// implementation
int area (int x1,int y1,int x2, int y2){
   return length(x1,y1,x2,y1) *
          length(x1,y2,x1,y2);
}
...
```

makefile names

`make` looks automatically for : `makefile, Makefile`

Override by using –f :

```
make –f MyMakefile
```

Makefile - version 1

A very simple Makefile

```
prog:
	gcc -Wall square.c main.c -o prog
```

Beware of the essential **tab**!

This is what you would type to compile and link the program

Makefile - version 2, macros

Macros are similar to variables
Upper case by convention

```
CC = gcc
CCFLAGS = -Wall

prog:
    $(CC) $(CCFLAGS) square.c main.c -o prog
```

We still run the same terminal command... because there are **no dependencies for prog**

Makefile - Version 3 - using dependencies

```
CC = gcc
CCFLAGS = -Wall

prog: square.o main.o
      $(CC) square.o main.o –o prog

main.o: main.c square.c square.h
      $(CC) $(CCFLAGS) -c main.c

square.o: square.c square.h
      $(CC) $(CCFLAGS) -c square.c
```