Modular Programming

Part 3: Modules, Include Files, Macros, Make Utility

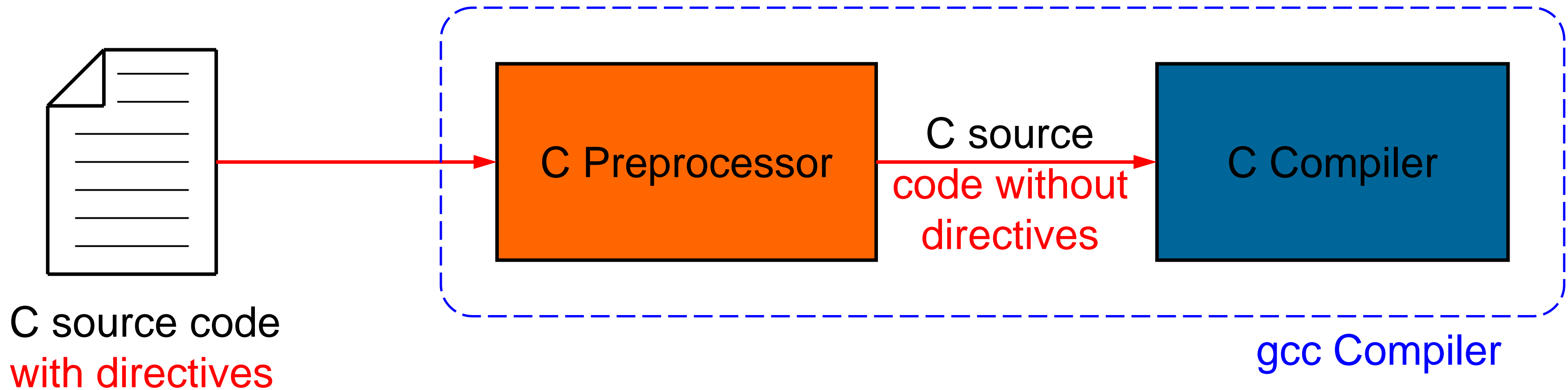
C Language Tutorial
System Programming 251-0053
Winter Semester 2005/06

René Müller, IFW B45.2

Content

- C Preprocessor
- Macros
- Splitting a program into several modules
- Header files
- Object files
- Libraries
- GNU Make-Utility

C Preprocessor and Directives



Directives are instructions for the preprocessor. They can contain complex statements, such as if-then-else. By using compiler directives meta programs can be written. The preprocessor language are interpreted by the preprocessor and removed before fed to the C compiler.

In the sequel we look at a few of this preprocessor instructions.

#define

#define declares a macro symbol. Every occurrence of this symbol later on will be replaced by the value of the symbol. Note this is a purely textual substitution.

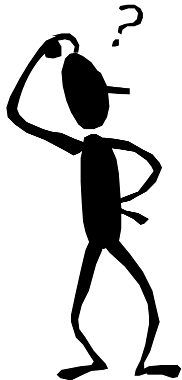
without semicolon!

Example:

```
#define NUMBER_OF_SENSORS   3
...
int sensors[NUMBER_OF_SENSORS];
```

is replaced by the preprocessor to:

```
int sensors[3];
```



Question: Why not using a constant, i.e., the following code?

```
const int NUMBER_OF_SENSORS = 3;
...
int sensors[NUMBER_OF_SENSORS];
```

#define Directive (2)

Macros may improve legibility of a C program:

```
#define LMOUSEBUTTON_PRESSED  (bttn & 0x0008)
...
if (BUTTON_PRESSED) {
   ....
}
```

is replaced by the preprocessor with

```
if ((bttn & 0x0008)) {
   ...
}
```

This example illustrates the advantage of using #defines.
`BUTTON_PRESSED` is more intuitive than **`btttn & 0x0008`**.

Preferred in system programming, close to the hardware level.

Cool stuff with defines (3)

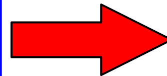
#defines are very powerful. Basically by the use of define the language can be altered. As the following example illustrates.

```
#include <stdio.h>

#define PROGRAM int
#define BEGIN main() {
#define END }
#define WriteLn printf

PROGRAM Hello;

BEGIN
    WriteLn(“Hello World“);
END;
```



```
#include <stdio.h>







int Hello;

main() {
    printf(“Hello World“);
};
```

The program is almost a valid Pascal Program. Difference: In Pascal strings are delimited by single quotes ‘ and last END has to be followed by a dot instead by a semicolon.

Macros with parameters

Macros can also take parameters. Remember: Macro are not subprograms. In the "invocation" statements they are textually replaced by the preprocessor.

Example:

```
#define MAX(a,b)  (((a)>(b))?(a):(b))

int x,y,z;
...
z = MAX(x+y,x+z);
...
```

Parentheses around operands a, b are sometimes necessary. (not in this case). Used for explicitly specifying precedence, if expressions are used as operands, e.g., x+y.

Defines do not need to have a "value", i.e., can be empty:

```
#define DEBUG
```

Defines an empty symbol DEBUG (can be used in #if later)).
Mit **#undef** removes symbol definition.

Define-Statements are usually placed at the beginning of a C file.

More cool stuff with #define

Defines can completely mix up things:

```
#define switch int
#define if while
#define float for
#define void(x) printf(x)

main() {
   switch a=1, b, c=3;

   if (a<c) {
       float(b=0;b<2;b++) {
          void("ugly");
       }
       a++;
    }
}
```

```
main() {



}
```

Quiz: what is the resulting program that finally will be compiled by the C compiler? What is its output?

#ifdef, #ifndef, #else und #endif Directives

If-Statements started with #ifdef and #ifndef und finished by #endif. Optionally an #else can appear between #if and #endif

Statements inside #ifdef are used, if the specified symbol was defined earlier.
Statements inside #ifndef (if-not-def) are used if, the specified symbol was **NOT** defined earlier.

Example: Hint for branch unit is compiler specific. Since the following is only valid for GCC its definition has to wrapped inside an #ifdef.
(from linux sources linux/compiler.h):

```
#if __GNUC__ > 3
   #define expect(foo,bar) __builtin_expect((long)(foo),bar)
#else
   #define __builtin_expect(foo,bar) (foo)
   #define expect(foo,bar) (foo)
#endif
#define __likely(foo) expect((foo),1)
#define __unlikely(foo) expect((foo),0)


                                 // on PowerPC
if (__unlikely(x!=0)) ... →  beq+ cr7,.L2
if (__likely(x!=0))   ... →  beq- cr7,.L2
```

#ifdef, #ifndef, #else und #endif Directives (2)

Example: DEBUG macro enables debugging output:

```
main() {
   int a=52, b=24;
   while (a!=b) {
      if (a>b) a=a-b;
      else b=b-a;
#ifdef DEBUG
      printf(a=%d, b=%d\n",a,b);
#endif
   }
   printf("gcd: %d\n", a);
}
```

If DEBUG “flag” is set during compilation the printf statement is included.

Thus the content of the two variables is written to stdout every loop iteration for debugging.

The final version (release) is compiled without flag DEBUG set.

DEBUG can either be set by ‘#define DEBUG‘ at the beginning of the C file.
Or even better – without any changes in the source code – on the command line when the compiler is invoked.

```
gcc -DDEBUG -o ggt ggt.c
```

Debugging-Version

```
gcc -o ggt ggt.c
```

Release-Version

Splitting Programs into Modules

Complex programs should be split into several reusable modules:

```
#include <stdio.h>
/* Prototypes */
int foo(int a, int b);
int bar(int a, int *result);

main() {
   int myVar=2,yourVar=3,res;
   res = foo(myVar, yourVar);
   printf("%d\n", res);
   bar(myVar, &res);
   printf("%d\n", res);
}

int foo(int a, int b) {
   return a+b;
}

void bar(int a, int *result) {
   *result = a;
}
```

```
#include <stdio.h>

main() {
   int myVar=2,yourVar=3,res;
   res = foo(myVar, yourVar);
   printf("%d\n", res);
   bar(myVar, &res);
   printf("%d\n", res);
}
```

testmodule.c

```
int foo(int a, int b) {
  return a+b;
}

void bar(int a, int *result) {
  *result = a;
}
```

mymodule.c

Splitting Programs into Modules (2)

Problem: Module testmodule.c must know prototypes of foo and bar.

Solution 1 (bad): Manually insert prototypes in main(). Note a whenever signature of a function module mymodule changes the prototype declaration of all C files that use this module have to be changed!

Solution 2 (good): Store Prototypes in a separate file mymodule.h (H means Header). All modules that use mymodule then must **include the content** of this header file, at the beginning of the file.

```
/* This is a header file. */
/* Prototypes */
int foo(int a, int b);
void bar(int a, int *result);
```

mymodule.h

#include Directive

With #include the preprocessor is asked to replace the token „**`#include filename`**“ **by the content** of the file „**`filename`**“.

For Module testmodule it looks like this:

```
#include "mymodule.h"

main() {
   ...
}
```

mymodule.c

after preprocessor

```
/* This is a Header file. */
/* Prototypes */
int foo(int a, int b);
void bar(int a, int *result);

main() {
    ...
}
```

Where are these header files located? There are two variants:

```
#include <gl/opengl.h>
```

Names between `< >` are looked up in the header directory (e.g., /usr/include). Further directories can be specified when the compiler is invoked (e.g., **`-I.`**, **`-I../../include`**).

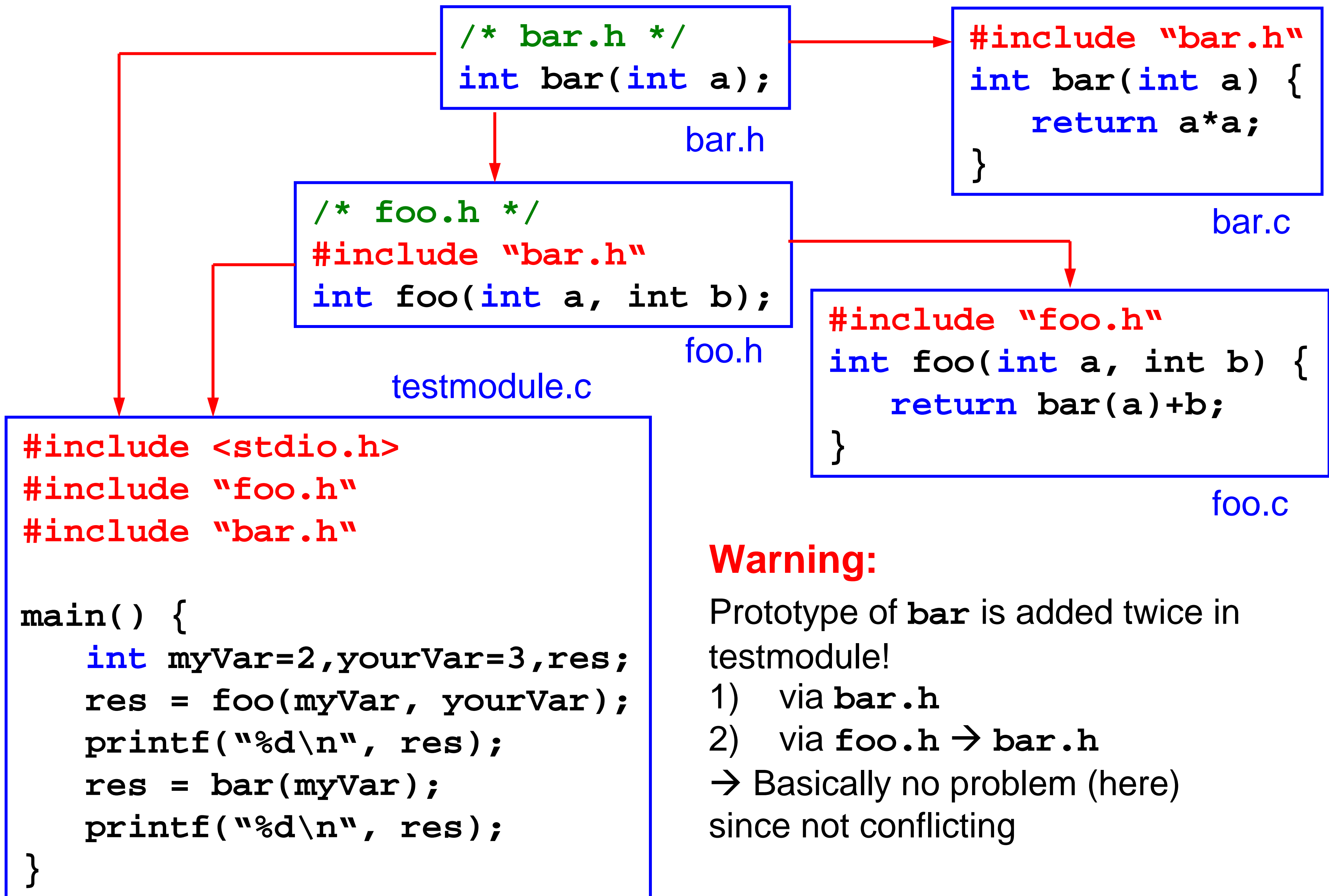
```
#include "mymodule.h"
```

Files between **" "** are looked up relative to the current directory.

Header Files

- Header files can contain
  - Function prototypes
  - Type definitions (structs, unions, enums, typedefs)
  - (Definition of classes in C++)
  - #define macros
  - #pragma instructions for the compiler
  - global variables
    - Creates a global variable in every module that includes the H file unless variable is declared **`extern`**.
      - Name clash
  - Implementation of inline functions
    - Calls of inlined functions are directly replaced by the body of the inline function

Multiple Includes



Cyclic Dependencies

- Compilation loops forever!
  - gcc –c vector.c

vector.h

```
#include "matrix.h"

typedef struct {
    double *v;
    int n;
} vector;

void add(vector *a,
  vector *b, vector *c);
void mmult(matrix *a,
  vector  *x, vector *y);
...
```

matrix.h

```
#include "vector.h"

typedef struct {
    int n;
    int m;
    double **A;
}

void eig(matrix *m, vector *v);
...
```

vector.c

```
#include "vector.h"
...
```

Preventing Multiple Includes

Question: How can multiple inclusions be avoided?

Answer: Use `#define` and `#ifndef` such that content is only included the first Time when the #include is reached.

Header file **foo.h**:

```
#ifndef _FOO_H_
#define _FOO_H_

#include "bar.h"

int foo(int a, int b);

#endif  // _FOO_H_
```

Header file **bar.h**:

```
#ifndef _BAR_H_
#define _BAR_H_

int bar(int a));

#endif  // _BAR_H_
```

Idea: Only if `_FOO_H_` and `_BAR_H_` are not yet defined, the content between `#ifndef` and `#endif` is inserted. At the same time the symbol is defined, such that further inclusions are prevented. → Gets rid of multiple inclusions.

Compiling Several Modules

Single modules, i.e., C files can be compiled into object files when the command line argument “-c” is specified.

```
$ gcc -c foo.c
$ gcc -c bar.c
$ gcc -c moduletest
```

creates object files (.o)

```
$ gcc -o moduletest *.o
```

creates executable

Note: There is a dependency between header files and the C files that include these headers, i.e., if a H file changes the dependent modules have to be recompiled.

If the project consists of thousands of source files the whole compilation process with all files can take a long time.

However it is also very time-consuming figuring out which modules have to be recompiled. If some are forgotten, the linker may throw errors if it tries to use an old object file or worse if the signature does not change the error will not be detected until sometime at runtime.
Therefore a tool/utility is needed that takes care of rebuilding stalled object files. ➔ Make utility.

Object Files

- .c (and .s) source files are processed by the compiler (assembler) into object files
- Example:

```
/* globals */
int v;
int w = 42;
extern int x;
static int y;
static int z = 7;

int main(int argc, char **argv)
{
    foo();
    y = x;
    printf("Hello World!\n");
    return 0;
}
```

- `$ gcc -c file.c`
- `$ nm file.o`

```
         U foo
00000000 T main
         U printf
00000004 C v
00000000 D w
         U x
00000000 b y
00000004 d z
```

helpful utility to resolve "undefined reference" problems

- Utility **nm** lists symbols from object files:

U: symbol is undefined

global:
T: symbol is in text section
C: symbol is common (uninitialised data)
D: symbol initialised in data section

local:
b: local symbol in uninitialised data section (bss)
d: local symbol in initialised data section

visibility of symbol in other modules

External Declarations in Modules

- Module `foo.c` with "exported" variable **common**.

```
/* module foo.c */
int common = 42;

int foo() {
    ...
    common++;
}
```

- `$ gcc -c foo.c`
- `$ nm foo.o`

```
00000000 D common
00000000 T foo
```

symbol common **defined** in data section (initialised)

- Module bar.c "imports" variable **common**.

```
/* module bar.c */
extern int common;

int bar() {
    ...
    common--;
}
```

- `$ gcc -c bar.c`
- `$ nm bar.o`

```
00000000 T bar
         U common
```

symbol **undefined** in this module

External Declarations in Modules (2)

- Module `foo.c` with "exported" variable `common`.

```
/* module foo.c */
int common = 42;

int foo() {
    ...
    common++;
}
```

- `$ gcc -c foo.c`
- `$ nm foo.o`

```
00000000 D common
00000000 T foo
```

symbol common **defined** in data section (**initialised**)

- Module bar.c "exports" variable `common` as well but **UNINITIALISED**!

```
/* module bar.c */
int common;

int bar() {
    ...
    common--;
}
```

- `$ gcc -c bar.c`
- `$ nm bar.o`

```
00000000 T bar
         C common
```

symbol common **defined** but **uninitialised** in bss section

⇒ **Works, because one is initialised and the other not.**

⇒ **At the end there is only a single variable `common` used by both**

External Declarations in Modules (3)

- Module `foo.c` with “exported” variable `common`.

```
/* module foo.c */
int common = 42;

int foo() {
    ...
    common++;
}
```

- `$ gcc -c foo.c`
- `$ nm foo.o`

```
00000000 D common
00000000 T foo
```

- Module bar.c “exports” variable `common` **INITIALISED**.

```
/* module bar.c */
int common = 1;

int bar() {
    ...
    common--;
}
```

- `$ gcc -c bar.c`
- `$ nm bar.o`

```
00000000 T bar
         D common
```

$\Rightarrow$ **Does not work! (Inconsistent initialisation)**

$\Rightarrow$ **Linker: “multiple definitions of symbol ‘`common`’”**

Static Symbols

- Sometimes you only want a global variable inside your module that does not need to be exported.
- **Declare the symbol as static.**
- Module `foo.c` with static global variable `common`.

```
/* module foo.c */
static int common = 42;
int foo() {
   common++;
}
```

- `$ gcc -c foo.c`
- `$ nm foo.o`

```
00000000 d common
00000000 T foo
```

symbol locally **defined** in data section (**initialised**)

- Note: **`static`** and **`external`** are not limited to variables. They apply to all symbols, thus also to functions.
- Module bar.c with static global variable `common`.

```
/* module bar.c */
static int common = 1;
int bar() {
   common--;
}
```

- `$ gcc -c bar.c`
- `$ nm bar.o`

```
00000000 T bar
         d common
```

symbol locally **defined** in data section (**initialised**)

⇒ **Works, because each module has its independent variable common.**

Static Variables defined in Functions

- You can also declare local variables (defined in functions) **`static`**.
- Semantics: Static variables are initialised only once (the first time) and keep their value between invocations of the same function. But the variables are still local, i.e., theyc an only be accessed in the function they are defined in.
- Example: Count invocations of function foo()

```
int foo() {
    static int invocations = 0;
    ...
    invocations++;
}
```

- Local variable is allocated as static symbol in the data section

```
$ nm foo.o
00000000 T foo
00000000 d invocations.0
```

Libraries

- Creating libraries
  - A number of object files can be collected into a single static library
  - Tool: **ar <archivefile> <memberfile>**
  - Example

```
$ ls
bar.o foo.o foobar.o
$ ar rcs libmy.a bar.o foo.o foobar.o
$ ls
bar.o foo.o foobar.o libmy.a
```

packs object files and creates library file libmy.a. (option rcs: replace existing, do not warn when creating new archive, write index)

  - Library files usually have prefix “lib” and for static libraries extension “.a”. (shared and dynamically loaded libs have “.so”)

- Linking against libraries
  - **$ gcc -o myprog myprog.c libmy.a**

or

  - **$ gcc -o myprog myprog.c -L. -lmy**

Name of library (note: without prefix “lib”)

add current directory “.” to library search path

Libraries – Example

- C-Math Library
  - Provides general mathematic functions, e.g. **`sin(..)`**, **`cos(..)`**, **`sqrt(..)`**
  - Function prototypes defined in Header file **`math.h`**
  - Implementation in library **`libm.{a,so}`** (**`-lm`**, **`-L`** option is not needed since this library in the default search path)
- Example:

```
#include <math.h>
main() {
   double x = 3.14;
   x = sin(x);
}
```

- Compilation & Linking
  - $ gcc –o sinpi sinpi.c –lm

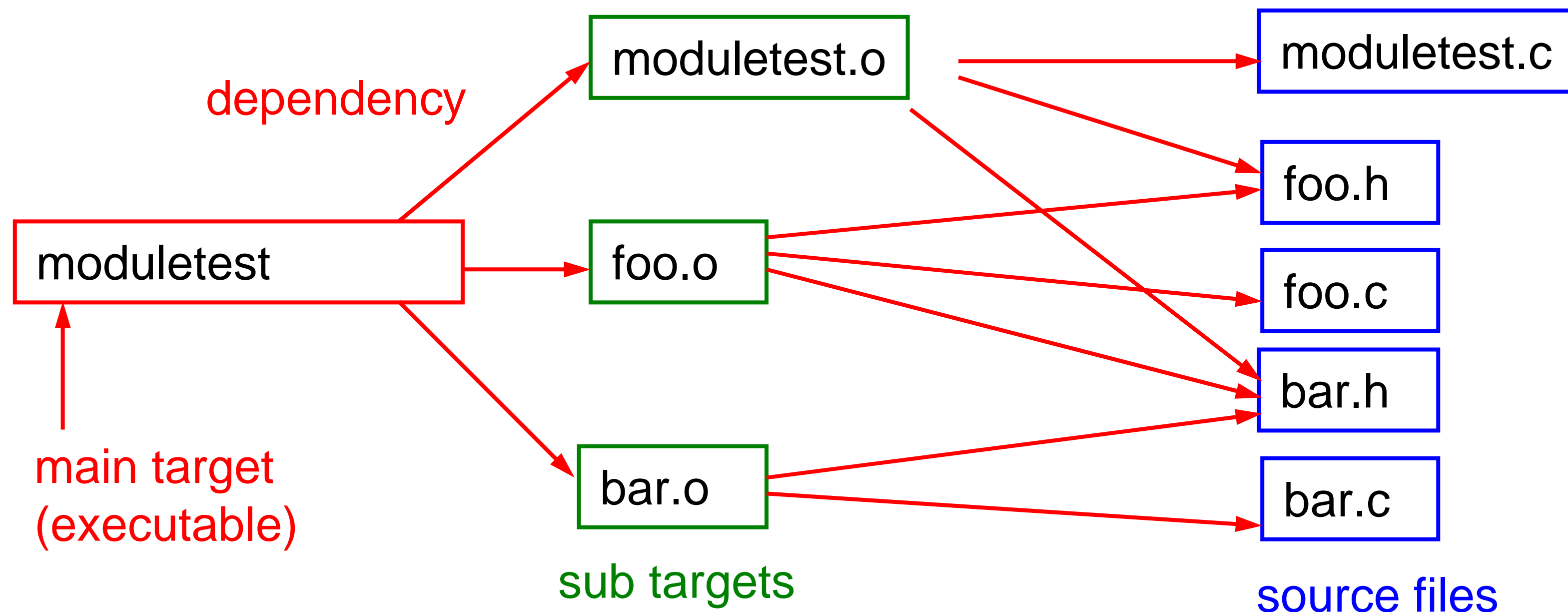
GNU Make Utility*

Make-Utility is a tool that can be used to automate a build process (not only for C/C++ but also for e.g. LaTeX). In general, it provides more flexibility than Integrated Development Environments such as MS Visual .NET or Borland C++.

Make-Utility uses a such called Makefile (file name '`Makefile`') that describes the dependencies in the project.

Make can be used to take care of stalled object files, i.e., make uses modification timestamps from the file system an rebuilds a target if the timestamp a target file is older than any of its source files.

The make file lists a set of targets (goals), the requirements to build that target and the generation instruction that creates the target from the sources.



*) Note: There are minor differences between GNU Make and BSD Make

The Makefile

This is the corresponding Makefile:

```
# This is a comment in my Makefile

moduletest: moduletest.o foo.o bar.o
	gcc -o moduletest moduletest.o foo.o bar.o

moduletest.o: moduletest.c foo.h bar.h
	gcc -c moduletest.c

foo.o: foo.c foo.h
	gcc -c foo.c

bar.o: bar.c bar.h
	gcc -c bar.c

clean:
	rm -f *.o
```

first rule is main target

additional target that can be explicitly in-voked in order to remove all .o files

→| = 1x Tab stop, i.e., <TAB> key

Important: „Blanks“ (<SPACE> key) do not work!

Use of Make

An rule in a make file has the general layout:

```
Target_Name: {Prerequisites ...}
	{Instruction to build this target}
```

When invoking make from command line the target can be specified which needs to be built. Make then also builds all required sub targets.

`$ make moduletest.o` Creates moduletest.o only

If make is invoked without command line option, make uses the default target, i.e., the first target specified in the Makefile.

`$ make` Creates foo.o, bar.o, moduletest.o and moduletest

In the previous example there is an additional target clean, that does not have any dependencies, since it only removes the .o files. It is usually very convenient to have it in the Makefile.

`$ make clean` Cleans up .o files.

Quirks with Make

- Variables
  - Variables can be defined as **`var_identifier = content`**
  - Content of variables can be used anywhere as **`$(var_identifier)`**
    Example:
```
CC = icc
foo.o: foo.c
       $(CC) -c foo.c
bar.o: bar.c
       $(CC) -c bar.c
```
  - Automatic variables
    - **`$@`** File name of the target of the rule
    - **`$<`** Name of the first prerequisite
    - **`$?`** Names of all prerequisites that are newer than the target
    - **`$^`** Names of all prerequisites
    - Example:
```
foo: foo.c
   gcc -o $@  $^
```
- PHONY Targets
  - A phony target is one that is not really the name of a file.
  - Avoid conflict with the a real file that accidentally has the same name
  - Example: target clean, what if there is a file with name “clean”?
    → If there is such a file, make will never execute this target! Thus specify:
  -
```
.PHONY: clean
clean:
        rm -f *.o
```

Adding Static Pattern Rules

- In general make should know how to build an .o file out from a .c file, i.e., you do not have to add a target for each of your 1000 C files. → Specify pattern that can be used for multiple targets
- Syntax:

```
targets ...: target-pattern: prereq-patterns ...
         commands
```

- Example:

```
CC = gcc            # compiler to use
CFLAGS -g -O3       # compiler flags

objects = foo.o bar.o … another 998 object files

all: $(objects)

$(objects): .o%: %.c
    $(CC) -c $(CFLAGS) $<  -o $@
```

Targets the rule applies to

Summary

- The C Preprocessor translates compiler directives and creates „pure“ C code that will be compiled by the C compiler.
- #include „filename“ is replaced by the content of the file. Allows definition of symbols for constants, expressions, macros, function prototypes in a central location available for use in several modules.
- #define defines macros
- #undef removes symbols defined by #define
- #ifdef, #ifndef, #else and #endif can be used to hide certain parts of the program.
- Complex software should be split into several modules.
- Header files contain interfaces of modules
- External dependencies are resolved during linking (statically) or loading (dynamically).
- Make utility can automatically (re-)build object files and executables while considering dependencies.
- Dependencies are defined in Makefile
- A Makefile must contain tabs in front of a command declaration