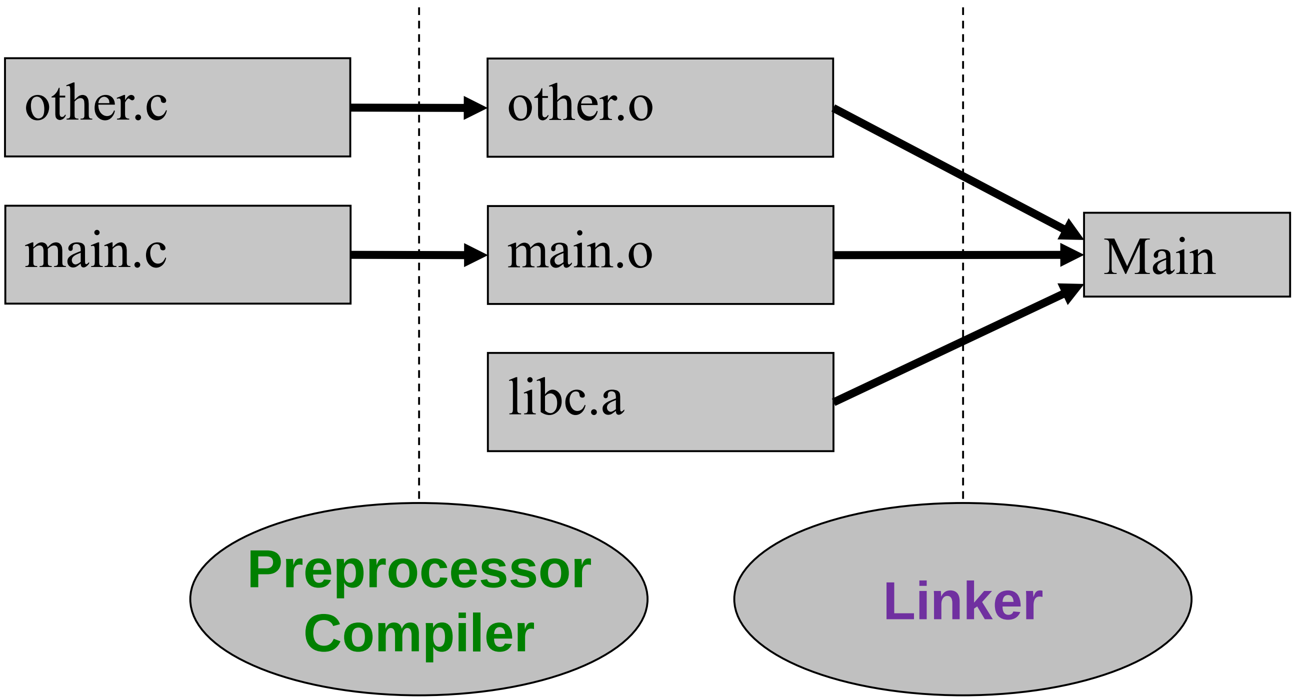
Makefile - Multiple file project management

©Copyright to Assaf Hoogi, computer Science Department

Process in Linux – longer way

(.c → .o → .exe)

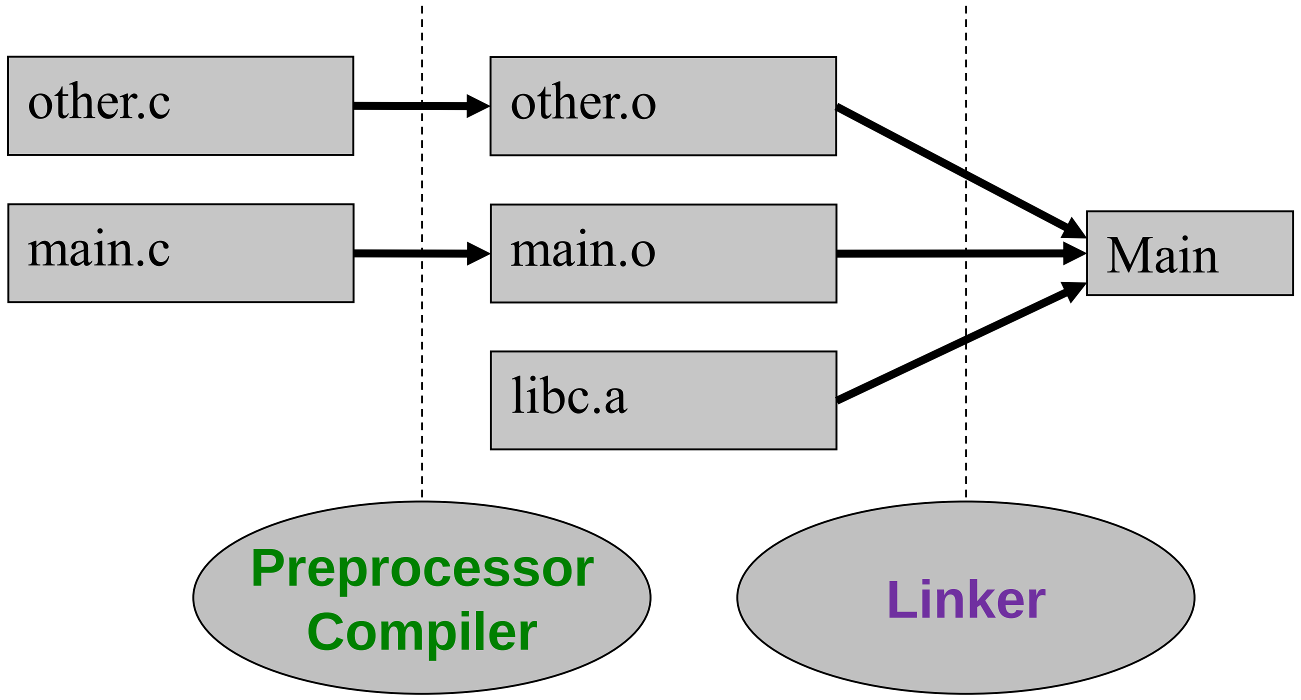
- $ **gcc –c** other.c –o other.o
- $ **gcc –c** main.c –o main.o
- $ **gcc** main.o other.o –o Main.exe



Process in Linux – shorter way

(.c → .exe)

- $ **gcc** main.c other.c –o Main.exe



Additional gcc commands

- $ gcc source.c - will compile the *source.c* file and give the output file as *a.out* file which is default name of output file given by gcc compiler, which can be executed as **a.exe**
- $ gcc source.c –o opt - will compile the source.c file but instead of giving default name, it will give output file as opt. -*o* is for output file option.
- $ gcc source.c -Werror -o opt - will compile the source and show the warning if any error is there in the program, -W is for giving warnings.
- $ gcc source.c -Wall -o opt - will check not only for errors but also for all kinds warning like unused variables. It is good practice to use this flag while compiling the code.

Additional gcc commands

- $ gcc -ggdb3 source.c -Wall -o opt - gives us permissions to debug the program using gdb3 which will be described later, *-g* option (that comes before gbd3) is for debugging.
- $ gcc -Wall source.c -o opt –lm - links the aux *math.h* library to our source file, -l option is used for linking particular library, for math.h we use -lm.

Make function for efficient build

- How can we compile a project with many files at once?
- What should we do if one file has been changed?
- **Make** function will solves the above problems

make and makefile

- make is typically used to build executable programs and libraries from source code. Generally speaking, make is applicable to any process that involves executing arbitrary commands to transform a source file to a target result.
- For example, make could be used to detect a change made to the source code
- It contains rules that the user should define
- The result - it can handle large number of files in an elegant way
- http://www.gnu.org/software/make/manual/make.html

make and Makefile

- Make is the command (in the linux screen) that run the Makefile
- Makefile is a rules-based programming
- Make will execute the first target's rules.
- If we want to execute the rules for the second one – we should explicitly mention that

Makefile

```
output1: dependency1
        rules1
output2: dependency2
        rules2
```

Make (the Linux command to run Makefile)

$make - (will run the rules for output1 only)

$make output2 - (will run the rules for output2 only)

Makefile syntax

- Makefile listing the rules for building the executable file.
- This is required only once, unless new modules are added to the program, and then the Makefile must be updated to add new module dependencies to existing rules and to add new rules to build the new modules.
- General syntax

  target : source1 source2 ... sourceN

  command

- prog1 : file1.c file2.c file3.c

  gcc -o prog1 file1.c file2.c file3.c

  this is a single tab - NOT a space (will not work with spaces)!!

- Goal of this makefile – take all these .c files and compile them to be .o files

Makefile process rules

- The general structure of the makefile –

  output: dependency

  rules

- prog1: read.c main.c list.c

  gcc –o prog1 main.c read.c list.c

- Here - .c files (dependencies / sources) are compiled them to .o files
- If prog1 **does not exist** - it will be created during the process
- If prog1 **does exist** , then its timestamp will be compared to the timestamps on main.c, read.c, and list.c. If any of the sources have been modified since prog1 was created, gcc will run to update prog1.
- If any of the sources was the target of another rule (e.g. main.o), then the timestamps on ***its*** dependencies would be checked recursively and rebuilt before rebuilding prog1

Example

What's the purpose of this makefile ?

```
Final_prog : foo.o bar.o
        gcc -o Final_prog foo.o bar.o
foo.o: foo.c
        gcc -c foo.c
bar.o: bar.c
        gcc -c bar.c
```

Generating Final_prog by compiling foo.o and bar.o

When you run ***make*** command, this makefile will run:

- foo.o and bar.o will be updated if needed.
- If Final_prog **<u>does exist</u>** , then its timestamp will be compared to the timestamps on foo.o and bar.o.
- If foo.o and bar.o have been modified since Final_prog was created, the gcc command will update Final_prog.

Another example

- Create the following makefile, which contains rules to build the executable, and save in the same directory as the source file. Use "tab" to indent the command (NOT spaces).

What's the makefile goal ?

```
all: hello.exe

hello.exe: hello.o
	gcc -o hello.exe hello.o

hello.o: hello.c
	gcc -c hello.c

clean:
	rm hello.o hello.exe
```

```c
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Makefile – Summary of key points

(20 min)

https://www.youtube.com/watch?v=i3tYp88YHbI&ab_channel=ProgrammingKnowledge2

Diving deeper - makefile content

- Explicit rules - explicit rules are instructions for specific files – as we saw until now
- Implicit rules - implicit rules are general instructions for files without explicit rules
- Variables/MACROS definitions
- Comments

Explicit rules

- A **rule** appears in the makefile and says when and how to remake certain files (**targets)**. It lists the other files that are the **prerequisites** of the target, and **commands** to use, create or update the target.
- Explicit rules specify the specific instructions to follow when we build specific targets. Explicit rules name one or more targets followed by (:) or by (::).
  - Single colon (:) means one rule is written for the target(s);
  - Double colons (::) mean that multiple rules are written for the target(s).

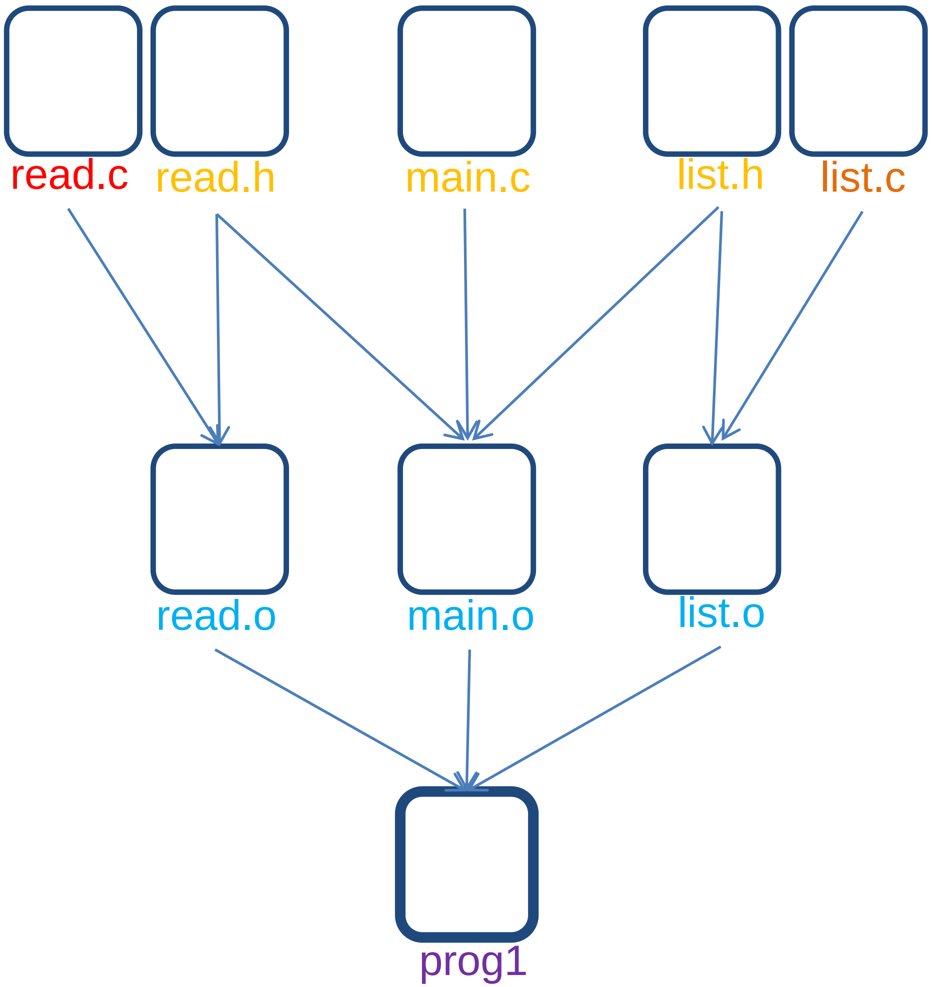
Explicit makefile

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



Comments

- Comments in makefile will come with #

Macros / Variables

- A macro is a variable that make expands into a string whenever make encounters the macro in a makefile.
- For example, you can define a macro called LIBNAME that represents the string "mylib.lib." To do this, type the line LIBNAME = mylib.lib at the beginning of the makefile. Then, when make encounters the macro $(LIBNAME), it substitutes the string mylib.lib. (Definition, Call)
- If make finds an undefined macro in a makefile, it looks for an operating system environment variable of that name and uses its definition as the expansion text.
- For example, if you wrote $(PATH) in a makefile and never defined PATH, make would use the text you defined for PATH in your AUTOEXEC.BAT. See your operating system manuals for information on defining environment variables.

Macros / Variables

אוניברסיטת אריאל בשומרון

```
<MacroName> = <expansion_text>
```

Element	Description
<MacroName>	Is case-sensitive (MACRO1 is different from Macro1). Limited to 512 characters.
<expansion_text>	Is limited to 4096 characters. Expansion characters may be alphanumeric, punctuation, or spaces.

- For example, if you define the following macro

SOURCE = f1.cpp f2.cpp f3.cpp

Declaration

You can substitute the characters .cpp for the characters .obj by using the following make command:

$(SOURCE:.cpp=.obj)

Use

- Rules for macro substitution:
  - Syntax: $(MacroName:*original_text*=new_text)
  - No space before or after the colon
  - Characters in *original_text* must exactly match the characters in the macro definition (text is case-sensitive)

Macros / Variables

- Macros make it easier to change one option throughout the file
- They also makes the makefile more reusable for another project
- In general it is a good idea to use variables to represent external programs. This allows users of the *makefile* to more easily adapt the *makefile* to their specific environment.

```
OBJFILES = file1.o file2.o file3.o
PROGRAM = myprog
CC = gcc
CFLAGS = -g –Wall

$(PROGRAM): $(OBJFILES)
	$(CC) $(CFLAGS) -o $(PROGRAM) $(OBJFILES)
```

Macros definition

What's the output of this makefile ?

Automated Macros

- *Automatic variables/macros* are set by *make* after a rule is matched.
- They provide access to elements from the target and prerequisite lists so you don’t have to explicitly specify any filenames (like built-in macros).
- They are very useful for avoiding code duplication, but are critical when defining more general pattern rules.

Core Macro	Macro’s role
$@	The filename representing the target
$%	The filename element of an archive member specification
$<	The filename of the first prerequisite / source
$?	The names of all prerequisites that are newer than the target, separated by spaces
$^	The filenames of all the prerequisites, separated by spaces
$*	The stem of the target filename. A stem is typically a filename without its suffix

1st Example

- What is this code below doing -

```
hello.o: hello.c hello.h
         gcc -c $< -o $@
```

- *hello.o* is the output
- The $< represents the first dependency (hello.c)
- The $ @ represents the output (hello.o)
- The -c flag compiles the files .
- The -o specifies the output file to create.

2nd Example

אוניברסיטת
אריאל
בשומרון

- The Makefile builds the hello executable if any one of main.cpp, hello.cpp, factorial.cpp changed (.cpp is c++ extension and g++ is a compiler. The idea is very similar to C, except for the extension that should be .c and we will use gcc compiler). The smallest possible Makefile to achieve that specification could have been:

```
hello: main.cpp hello.cpp factorial.cpp
    g++ -o hello main.cpp hello.cpp factorial.cpp
```

How can we improve efficiency ?

By separating the source files – let's try to write down the solution

Solution

```
OBJECTS=main.o hello.o factorial.o

hello: $(OBJECTS)
	g++ -o hello $(OBJECTS)

main.o: main.cpp
	g++ -c main.cpp

hello.o: hello.cpp
	g++ -c hello.cpp

factorial.o: factorial.cpp
	g++ -c factorial.cpp
```

Improve efficiency – by compiling only those C++ (or C) files that were re-edited.

Let squeeze the lemon ☺ and let improve it even further

Pattern Rules

אוניברסיטת אריאל בשומרון

- We can define an implicit rule by writing a *pattern rule*.
- A pattern rule looks like an ordinary rule, except that its target contains the character ‘%’.
- A pattern rule ‘%.o : %.c’ says how to make any toy file *toy.o* from another file *toy.c*.

```
CC=gcc
CFLAGS= -g -Wall
%.o: %.c
	$(CC) -c $(CFLAGS) $< -o $@
```

Conversion from .c to .o

Old-Fashioned **Suffix Rules**

- In older Makefiles, you may see rules like the one below, which have a similar effect:

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

- This means: “To create *filename*.o from *filename*.c, run gcc -c -g -Wall -c *filename*.c -o *filename*.o “

Solution 2

Here can we improve the previous solution by replacing all object file rules

with a single .cpp.o (suffix) rule:

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
    $(CC) $(CFLAGS) $< -o $@
```

Using Wildcards

אוניברסיטת אריאל בשומרון

- Makefile often contains long lists of files. To simplify this process make supports wildcards.
- Wildcards can be used in a target, prerequisite, or command context.
- Example wildcards :
  - * - The asterisk in a wildcard matches any character zero or more times. For example, "comp*" matches anything beginning with "comp" which means "comp," "complete," and "computer" are all matched.
  - ? - A question mark matches a single character once. For example, "c?mp" matches "camp" and "comp." The question mark can also be used more than once - for example, "c??p"
  - [] - Specifies one of any character in a set, as in `c[auo]t`, which locates "cat“, "cut“, and "cot". Anything not in that range like a number would not be matched.
  - ^ – Specifies one of any character not in the set, as in `st[^oa]ck`, which excludes "stock" and "stack" but returns "stick" and "stuck." The caret (^) must be the first character after the left bracket ([) that introduces a set.
- Wildcards can be very useful for creating more adaptable makefiles. For instance, instead of listing all the files in a program explicitly, you can use wildcards:

```
prog: *.c
        $(CC) -o $@ $^
```

Implicit rules

- An implicit rule specifies a general rule for how make should build files that end with specific file extensions.
- Implicit rules start with either a path or a period. Their main components are file extensions separated by periods.
- *Implicit rules* are mostly pattern rules or suffix rules found in the rules database built-in to make.

Implicit Rules

- Explicit rules so far, e.g:
-

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

- Implicit rules: “.o” files from “.c” files.

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

No need to tell *make* to create foo.o and bar.o from their .c equivalents

- If we would like to write this with a pattern rule (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)

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

Rules that don’t create their target

- A rule that creates its target

  myprog: file1.o file2.o file3.o

  gcc -o myprog file1.o file2.o file3.o

- This rule does not create a file named “clean”

  clean:

  rm file1.o file2.o file3.o myprog

- *make* assumes that a rule's command will build/create its target
- If the target (“clean”) is not present, make will execute the commands!
- If your rule does not actually create its target, the target will never exist, so the rule will execute its commands (e.g. clean above)
- make clean is a common convention for removing all generated files

Rules with no commands

```
all: myprog myprog2
myprog: file1.o file2.o file3.o
	gcc -o myprog file1.o file2.o file3.o
myprog2: file4.c
	gcc -o myprog2 file4.c
```

- The *all* rule has no commands, but depends on myprog & myprog2
- Typing $make all (in Linux worksapce) will ensure that myprog, myprog2 are up to date because their dependencies will be checked recursively and updated as needed
- Having an all rule is also a common convention, and is often put as the first rule in a file, so that just typing make without giving a target will build "everything" (the dependencies of all)

.Phony

- Sometimes we want that Makefile will run commands that do not represent physical files in the file system .
- Good examples for this are the common targets "clean" and "all". Chances are this isn't the case, but you may potentially have a file named clean in your main directory. In such a case Make will be confused because by default the clean target would be associated with this file and Make will only run it when the file doesn't appear to be up-to-date with regards to its dependencies.
- These special targets are called *phony* er'yeht ekaM llet ylticilpxe nac uoy dna selfi htiw detaicossa ton

```
.PHONY: clean
clean :
    rm -rf *.o
```

.Phony

- Let's assume you have install target, which is a very common in makefiles.
- If you do not use .PHONY, and a file named install exists in the same directory as the Makefile, then make install will do nothing. This is because Make interprets the rule to mean "execute such-and-such recipe to create the file named install". Since the file is already there, and its dependencies didn't change, nothing will be done.
- However if you make the install target PHONY, it will tell the make tool that the target is fictional, and that make should not expect it to create the actual file. Hence it will not check whether the install file exists, meaning: a) its behavior will not be altered if the file does exist and b) extra stat() will not be called.
- Generally all targets in your Makefile which do not produce an output file with the same name as the target name should be PHONY. This typically includes all, install, clean, distclean, and so on.

Libraries

Libraries

- Library is a collection of pre-compiled pieces of code that can be reused in a program. Libraries simplify life for programmers, in that they provide reusable functions, routines, classes, data structures and so on which they can be reused in the programs.

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

Static libraries (.a)

- General - static libraries are joined to the main module of a program during the linking stage of compilation before creating the executable file. After a successful link of a static library to the main module of a program, the executable file will contain *both* the main program and the library.
- How to create them - static libraries are created using some type of archiving software, such as **ar**. **ar** takes one or more object files (that end in .o), zips them up, and generates an archive file (ends in .a) — This is our “static library”.
- .lib is an example for static library

Shared/ dynamic libraries (.so)

אוניברסיטת אריאל בשומרון

- Shared libraries are linked dynamically by simply including the address of the library (whereas static linking is a waste of space).
- Dynamic linking links the libraries at the run-time. Thus, all the functions are in a special place in memory space, and every program can access them, without having multiple copies of them.
- .dll is an example for shared library

Static Vs. Shared

אוניברסיטת אריאל בשומרון

- **Static libraries**, while reusable in multiple programs, are locked into a program at compile time.
- **Dynamic,** or **shared libraries** on the other hand, exist as separate files outside of the executable file.
- **Recompilation** -
  - The **downside of static library** is that its code is locked into the final executable file and cannot be modified without a re-compile.
  - In contrast, the **upside of dynamic library** - can be modified without a need to re-compile.
- **Copies in memory –**
  - Because **dynamic libraries** live outside of the executable file, the program need only make one copy of the library's files at compile-time.
  - Whereas using a **static library** means every file in your program must have it's own copy of the library's files at compile-time.

Static Vs. Shared

- **File corruption –**
  - The **downside of using a dynamic library** is that a program is much more susceptible to breaking. If a dynamic library for example becomes corrupt, the executable file may no longer work.
  - **A static library**, however, is untouchable because it lives inside the executable file.
- **Copies in memory -**
  - The **upside of using a dynamic library** is that multiple running applications can use the same library without the need for each to have it's own copy.
- **Runtime speed -**
  - **Benefit of static libraries** - execution speed at run-time. Because it is already included in the executable file, multiple calls to functions can be handled much more quickly than a dynamic library's code, which needs to be called from files outside of the executable.

Static Vs. Shared

אוניברסיטת אריאל בשומרון

PROPERTIES	STATIC LIBRARY	SHARED LIBRARY
Linking time	It happens as the last step of the compilation process. After the program is placed in the memory	Shared libraries are added during linking process when executable file and libraries are added to the memory.
Means	Performed by linkers	Performed by operating System
Size	Static libraries are much bigger in size, because external programs are built in the executable file.	Dynamic libraries are much smaller, because there is only one copy of dynamic library that is kept in memory.
External file changes	Executable file will have to be recompiled if any changes were applied to external files.	In shared libraries, no need to recompile the executable.
Time	Takes longer to execute, because loading into the memory happens every time while executing.	It is faster because shared library code is already in the memory.
Compatibility	Never has compatibility issue, since all code is in one executable module.	Programs are dependent on having a compatible library. Dependent program will not work if library gets removed from the system .

Generate Static library (.a)

- *Compile*:

$ gcc -Wall -c ctest.c

- We have the object file(s), we can archive them and make a static library using ar.

$ ar –rc libmine.a ctest.o

A static library called "libmine.a" is created. The -rc options create the archive without a warning and replaces any pre-existing object files in the library with the same name.

- There are two options to include the library in a makefile

$gcc prog.c –L. –l*mine* -o *myprog*

  - **-L** says "look in directory for library files"
  - **.** (the dot after 'L') represents the current working directory (./path/to/library-directory for any other library)
  - **-l** says "link with this library file"
  - **mine** is the name of our library. We omitted the "lib" prefix and ".a" extension. The linker attaches these parts back to the library's name later.
  - **-o myprog** says "name the executable file myprog"

Generate shared library (.so)

- Compile our library source code into position-independent code (PIC) -

  $ gcc -c -Wall -Werror -fpic foo.c

- Creating a shared library from an object file:

  $gcc -shared –Wall , foo.o -o libfoo.so

  The -shared key tells the compiler to produce a shared object which can then be linked with other objects to form an executable.

- Add the library path

  Our library has to be shared dynamically during the linking stage with other programs, and to make it happen, we have add a path to the library to the **LD_LIBRARY_PATH** environment variable:

  export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH

  If it is in the current working directory, then we can use the **.** to add its path. Now the operating system is aware of where to look if some program will request a functionality from the library.

- Linking with a shared library

  The linker should know where to find libfoo. GCC has a list of places it looks by default, but our directory is not in that list. We need to tell GCC where to find libfoo.so. We will do that with the -L option. Here, we will use the current directory:

  $ gcc -Wall main.c –L. -lfoo -o FinalProg