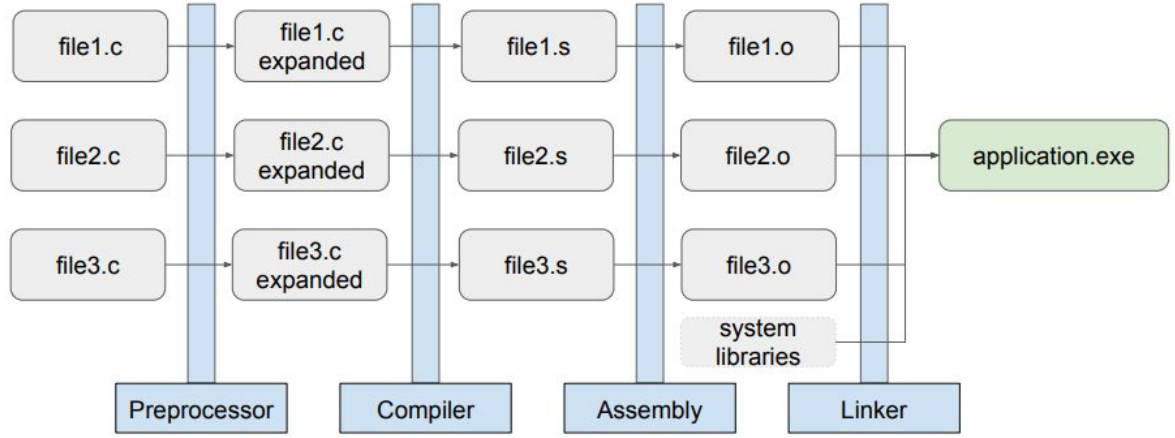
Continuous Integration

Build Automation

Build Automation - Compiling Using GCC

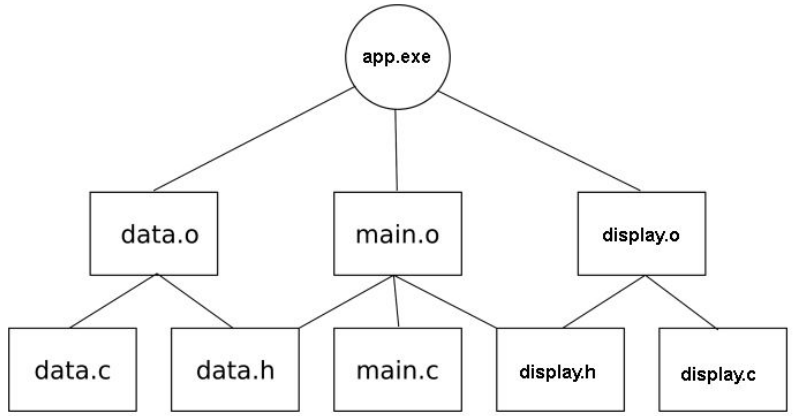
- Build process of a C program using gcc
  - **Preprocessor**: '#'-prefixed lines for includes, replacing macros, conditional compilation, et.c.
  - **Compiler**: Generate assembly code from the preprocessed code, checks the code for errors
  - **Assembler**: Makes machine instructions (object file) from the generated assembly code
  - **Linker**: Resolves symbols (function calls, global variables...) between software components and system libraries

- *gcc -E file.c -o file.i => Preprocessed code*
- *gcc -S file.c -o file.s => Assembly code*
- *gcc -c file.c => Object file (file.o is generated)*
- *gcc file1.o file2.o file3.o -o app => Linked file(app)*
- *gcc main.c -save-temps -o main*
- *gcc file1.c file2.c file3.c -o application*



Build Automation - Compiling Using GCC

- GCC has so many flags and options to control the compilation. 🔗 The GCC Manual
- Some useful options in the Preprocessor phase
  - gcc **-E** main.c -o main.i
    - Saves the preprocessed code in a **.i** file
  - gcc **-E -C** main.c -o main.i
    - Saves the preprocessed code in a **.i** file
    - Prevents the preprocessor from removing comments
  - **-D *name[=definition]*** to define a macro
  - **-U name** to undefine a macro defined on the command line or a GCC predefined macro. The **-D** and **-U** options are processed in the order in which they occur on the command line
  - **-I *directory*** is used to specify the directory path of header files.



A Simple Program

- gcc -c main.c
- gcc -c data.c
- gcc -c display.c
- gcc data.o main.o display.o -o app.exe

Build Automation - Compiling Using GCC

- *gcc* ***-S -fverbose-asm*** *main.c* generates *main.s* and includes variable names as comments in the assembly file
- Some useful options in the Assembling phase
  - *gcc* ***-c*** *main.c* generates *main.o* as the object file and
  - *gcc* ***-c -g*** *main.c* generates *main.o* for debugging
- Some useful options in the Linking phase
  - The standard library functions are usually in the file ***libc.a*** (the extension **.a** stands for **archive**) or in libc.so (the extension **.so** stands for **shared object**) in the default lib path
  - To link a non-standard library, **-l***library* for lib*library*.a shall be used
  - If the target supports shared libraries, lib*library*.so will be linked if **-static** option is not used
  - If the library is not in the default lib path, **-L***directory* shall be used to specify the library path
    - E.g. gcc -o test module.c test.c -lunity

Build Automation - Compiling Using GCC

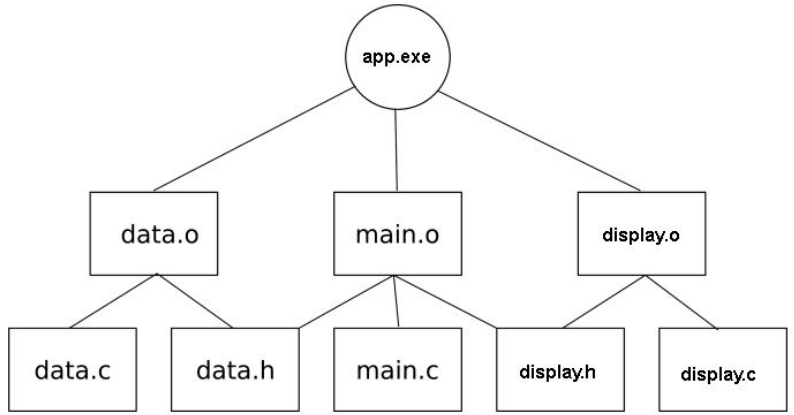
- gcc ***-fsyntax-only*** *main.c* only checks for correct syntax in main.c
- It is possible to compile multiple files in one command
  - ***gcc -c file1.c file2.c;*** it is equivalent to ***gcc -c file1.c*** and ***gcc -c file2.c***
  - But in a large program compiling all files in one command can cause problems
    - More memory is required and the compilation takes more time
- If the target supports shared libraries
  - A shared library is a special object file that can be linked to a program at runtime.
  - Using shared libraries has some advantages:
    - A program's executable file is smaller
    - Shared modules permit modular updating
    - Moore efficient use of the available memory.
  - gcc **-shared** -o lib<library-name>.so library-name.o creates a shared library
    - To compile the code use **-fpic** to generate position-independent code

Build Automation - Compiling Using GCC

- GCC supports different C versions using **-std=**xxx; e.g. gcc -std=c11 main.c
- **-pedantic-errors** used to fail on non-standard usage.
- **-pg** adds profiling information to programs for analysing
  - **gprof** generates a call graph and execution time of functions
  - E.g. gcc **-pg** main.c -o main && ./main && **gprof** ./main
- **-Wall** generates a wide range of warnings
- **-Werror** is used to fail on all warnings
- **-Wextra** generates warnings on legal but questionable usage.
  - E.g. check if an unsigned variable is lesser than zero
- **-Wconversion** generates warnings on implicit type conversions that change values

Build Automation - Make and Makefiles

- Manually compilation of a small program
  - gcc -c main.c (creates main.o)
  - gcc -c data.c (creates data.o)
  - gcc -c display.c (creates display.o)
  - gcc data.o display.o main.o -o app.exe



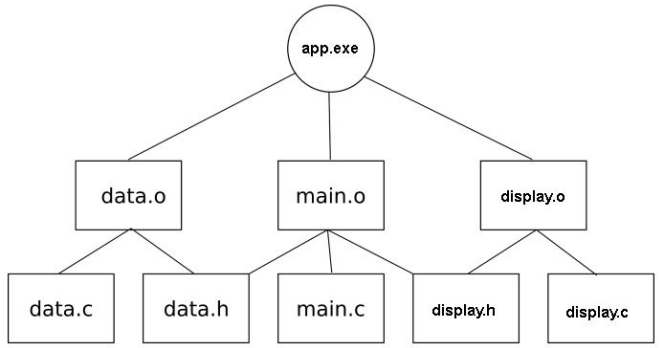
- It is clear we can not compile a large and complex program manually over and over
- **make** automates the process of building programs of any size and complexity
  - It manages the individual rules that define how to build various targets
  - It does not build unchanged files in projects by comparing files timestamps
  - It can build different parts of a large program and multiple execution units in one build
  - It uses special rules and syntax to build targets according their dependencies

Build Automation - Make and Makefiles

- The problem
  - T o make an executable program we need to link compiled object files
  - To generate object files, we need to compile C source files
  - The source files need to be preprocessed to include their header files.
  - If we edit a source or header file, then any file that was directly or indirectly generated from it needs to be rebuilt.
- The solution
  - **make** organizes the above tasks in the form of rules.
  - The make program has a special syntax for its rules.
  - ***make -f filename*** tells make which file contains the rules
  - Usually this option is omitted and **make** looks for a file with the default name **makefile** or **Makefile**.

Build Automation - Makefiles

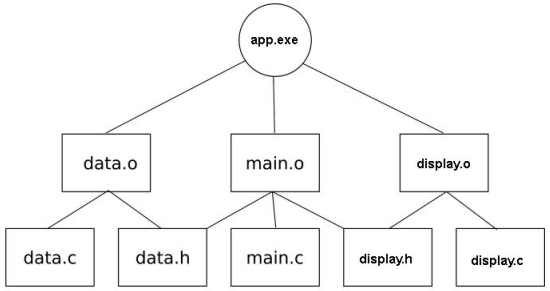
- General form of the rules in makefiles
  - **targets**: **dependencies**
    <TAB>**command**
    <TAB>**command**
    <TAB>**...**
    - Targets can be one or more targets
      - The first target must be placed at the beginning of the line, with no whitespace to the left of it
    - Dependencies can be zero or more dependencies
    - Each command line must start with a tab
- In addition to rules, makefiles also contain
  - Comments, variable assignments, macro definitions, include and conditional directives and functions.




```
# A basic Makefile

app.exe: data.o display.o main.o
    gcc -o app.exe data.o display.o main.o

data.o: data.c data.h
    gcc -c data.c

display.o: display.c display.h
    gcc -c display.c

main.o: main.c data.h display.h
    gcc -c main.c
```

Build Automation - Makefiles

- **make** parses Makefile and builds a dependency tree for targets
- Then uses the dependency tree to build the desired targets
  - A Makefile is not executed in sequential order
  - If a target is older than any dependencies (directly or indirectly)
    - **make** executes the commands.
  - A rule with **no** dependency tells only **how** to build the target, not **when** to build it

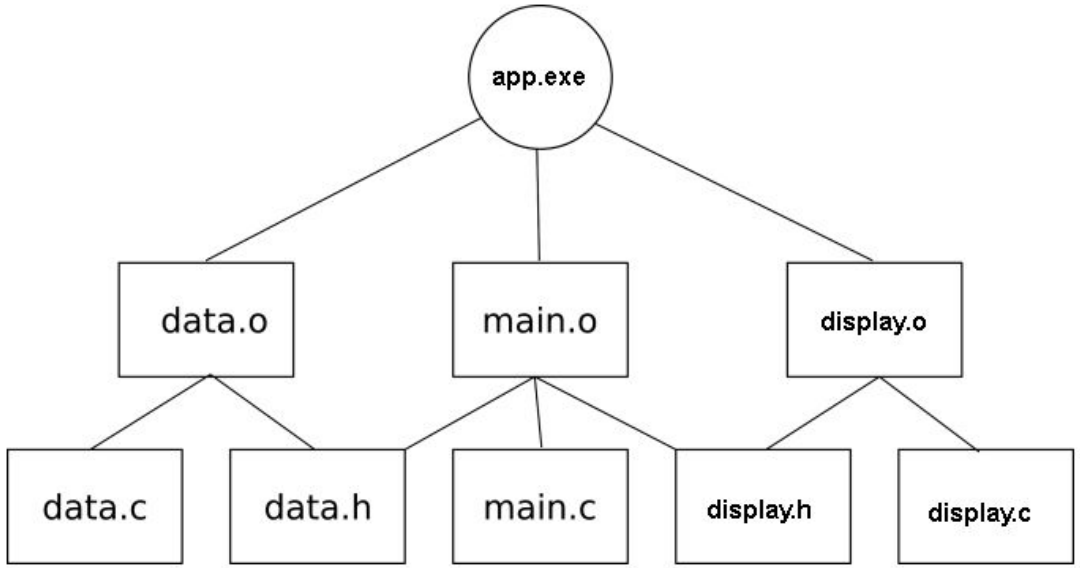


- Phony Targets: A target which is not name of a file
  - Just a name of a recipe to be executed
  - It is used to avoid a conflict with a file of same name
    - Phony targets are defined as .PHONY
    - E.g. clean, install, all and etc. To make: **make clean**

```
# A basic Makefile

.PHONY: clean mkbuild

clean:
    rm -rf build

mkbuild:
    mkdir -p build
```

Build Automation - Makefiles

- To build a target run ***make*** *target*. E.g. ***make clean***
  - ***make*** and ***make all*** are equivalent.
  - **all** is the top-level target the makefile knows about.



Makefile

```
# A basic Makefile

all: app

app: data.o display.o main.o
    gcc -o app.exe data.o display.o main.o

data.o: data.c data.h
    gcc -c data.c -o data.o

display.o: display.c display.h
    gcc -c display.c -o display.o

main.o: main.c data.h display.h
    gcc -c main.c -I./ -o main.o

.PHONY: clean all

clean:
    rm *.o
```

Build Automation - Makefiles

- **make** runs the commands in a rule individually in a separate shell instance
- Make sure that no command depends on the side effects of the previous command
- In the first rule, **rm** is not run in the **build** folder.

```
clean:
    cd build
    rm *.o
```

- To run several commands in the same shell
  - Put the commands in one line separated by semicolon
  - If you want to have more than one line use backslashes
  - Running multiple commands in the same shell can speed up processing

```
clean:
    cd build; \
    rm *.o
```

- In MinGW you need to fix make
  - Go to where MinGW has been installed (typically C:\\MinGW)
  - In **MinGW/bin/** make a copy of **mingw32-make.exe** in the same folder
  - Rename the copy to **make.exe**

Build Automation - Makefiles

- Using # you can comment
  - To comment between lines of a rule, **#** shall be in the beginning of the line
  - For **multiline** comments you can use **backslash** (\)
- In Makefiles we can use variables. Type of all variables is string
- Variable names can contain any character except :, = and #
  - But to be compatible with shell, you should use only **letters**, **digits** and **underlines**
  - To use variable they shall be enclosed in parentheses and prefixed with $.
  - E.g. **CC = gcc** and we can use CC like **$(CC)**
- When make applies a rule, it evaluates the variables used in the rule
- make evaluates variables by expanding them in two ways
  - ***recursively expanded*** or ***simply expanded***

Build Automation - Makefiles

- In recursively expanded variables
  - The reference of variables in the right side of assignments are stored, not their values.
  - **=** is used as the assignment operator
  - In the example y is recursive and the reference of $(x) is not expanded until y is evaluated.

```
x = hello
y = $(x)
# Both $(x) and $(y) will now yield "hello"
x = world
# Both $(x) and $(y) will now yield "world"
```

- In simply expanded variables
  - Variable references in the value are expanded immediately on assignment
  - The expanded values are stored, not references
  - **:=** s used as the assignment operator

```
x := hello
y := $(x)
# Both $(x) and $(y) will now yield "hello"
x := world
# $(x) will now yield "world", and $(y) will yield "hello"
```

- += operator is used to append more characters to the value of a recursively expanded variable. E.g. y = Hello and y += World!

Build Automation - Makefiles

- Variable ending substitution - to substitute ending substrings

  $(variable_name:ending=new_ending)

```
SOURCES = main.c data.c display.c
OBJECTS = $(SOURCES:.c=.o)
```

- It is possible to create and assign variables in the command line.
  - **make** [options] [variable_assignments] [targets separated by spaces]
  - E.g. ***make number=123 clean run***. Now number can be used in the Makefile
- Predefined variables in Makefiles
  - **$@** is the name of file being created by the rule.
  - **$*** is the prefix that is common to both target and dependent file
  - **$<** is the first dependency
  - **$^** is the list of dependencies.
  - **$+** is the list of dependencies, excluding duplicates.
  - **$?** is the list of dependencies that are newer than the target.

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

Build Automation - Makefiles

- Pattern Rules
  - Generalizing the rules to avoid explicit rules for each file
  - An pattern rule is where no specified target is given, but instead a wildcard (%)

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

- We can have macros in Makefiles. Macros are defined using **define** and **endef**
  - There is no difference between macros and variables
  - A macro is used like a variable. E.g. $(CLEAN)

```
define CLEAN
    @echo Clean The Project
    rm -rf build
endef
```

- **make** supports used-defined functions and has some built-in functions
- General form of calling a function is

  $(function_name arguments_list_separated_by_comma)

- **make** provides some useful functions
- User defined functions are like function-like macros

Build Automation - Makefiles

- A **user defined function** can be defined using a **macro** or a **variable**
  - Functions in **make** are variables with parameters references ($1, $2, $3, etc.)
- A user defined function can be called using the built-in function **call**.
  - The syntax of the call function is: **$(call variable,param,param,…)**
  - In the example **foo** contains **b a**

```
reverse = $(2) $(1)
foo = $(call reverse,a,b)
```

- Some useful built-in functions
  - *$(**subst** find_text,replacement_text,original_text)*
    - Is used to find and replace a ***find_text*** with ***replacement_text*** in ***original_text***
  - *$(**wildcard** pattern)*
    - Expands to a list of existing filenames that match the pattern. E.g. $(wildcard *.c)
  - *$(**patsubst** find_pattern,replacement_pattern,original_text)*
    - exchanging part of filename. E.g. ***objects := $(patsubst %.c,%.o,$(wildcard *.c))***

Build Automation - Makefiles

- **make** supports directives. Look at the directives
  - **define** and **endef** to define macros
  - Conditional directives: **ifeq**, **ifneq**, **ifdef**, **ifndef**, **else**, **endif**
  - **include** is used to a makefile in another one
    - **gcc -M**, **-MM** and **-MMD** generate the dependencies automatically
    - Similar to include in C but works differently
    - We can include multiple files in one include
      - E.g. **include** defaults.mk $(SRCDIR)/file.mk foo bar
    - Possible to use wildcards like * and ? in the file names
      - E.g. **include** defaults $(SRCDIR)/*.mk
  - **override** is used to change variables defined in command line. E.g. **make VAR=10 run** and in the Makefile we can have: **override VAR = 20**

```
OBJ = test.o fizzbuzz.o

ifdef SHAREDLIB
    LIB += unity.so
else
    OBJ += unity.o
endif

test: $(OBJ) $(LIB)
    $(CC) -o $@ $^
%.so : %.o
    $(CC) -shared -o $@ $<
```

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

Build Automation

- Some useful links
  - The GCC Manual
  - UPEvent: GCC and Makefiles
  - C Programming: Makefiles
  - GNU make Manual
  - A Shallow Dive into GNU Make
  - GCC and Make
  - Managing Projects with GNU Make
  - The GNU Profiler (gprof) Manual
  - How to use Linux GNU GCC Profiling Tool
  - GPROF - Unix, Linux Command