



# Writing and Testing

CS2263 – Systems Software Development



1

## Learning Outcomes

At the conclusion of this lecture students should be able to:

- List the phases of software development
- Discuss the role of structure and automation in the software development process
- Explain what the make utility does and build a simple makefile
- Explain what a dependency and target are, in the context of make



2

## References

- Lu, Yung-Hsiang. 2015. Intermediate C Programming. CRC Press. New York. Pp 9-27 (Chapter 5)
- Maguire, Steve. 1993. Writing Solid Code: Microsoft's techniques for developing bug-free C programs. Microsoft Press. Redmond, Wash.

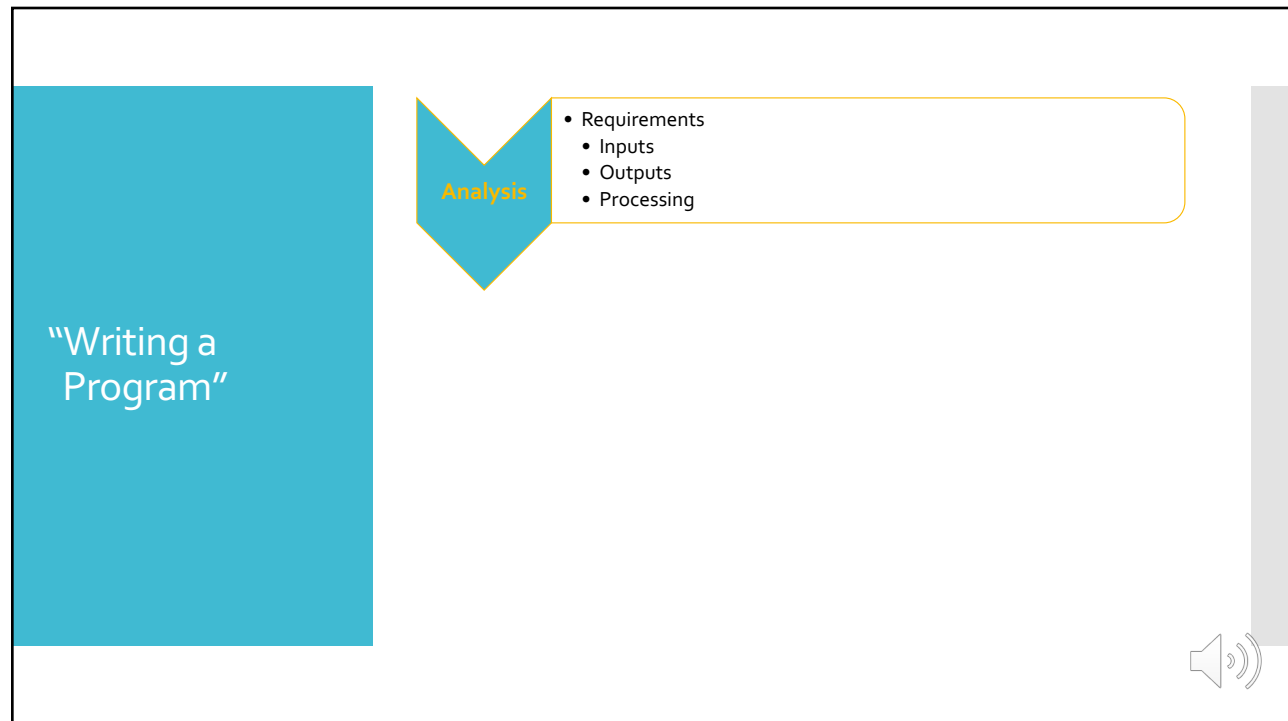


3

“You can’t change the plan  
if you haven’t got a plan  
to change”



4



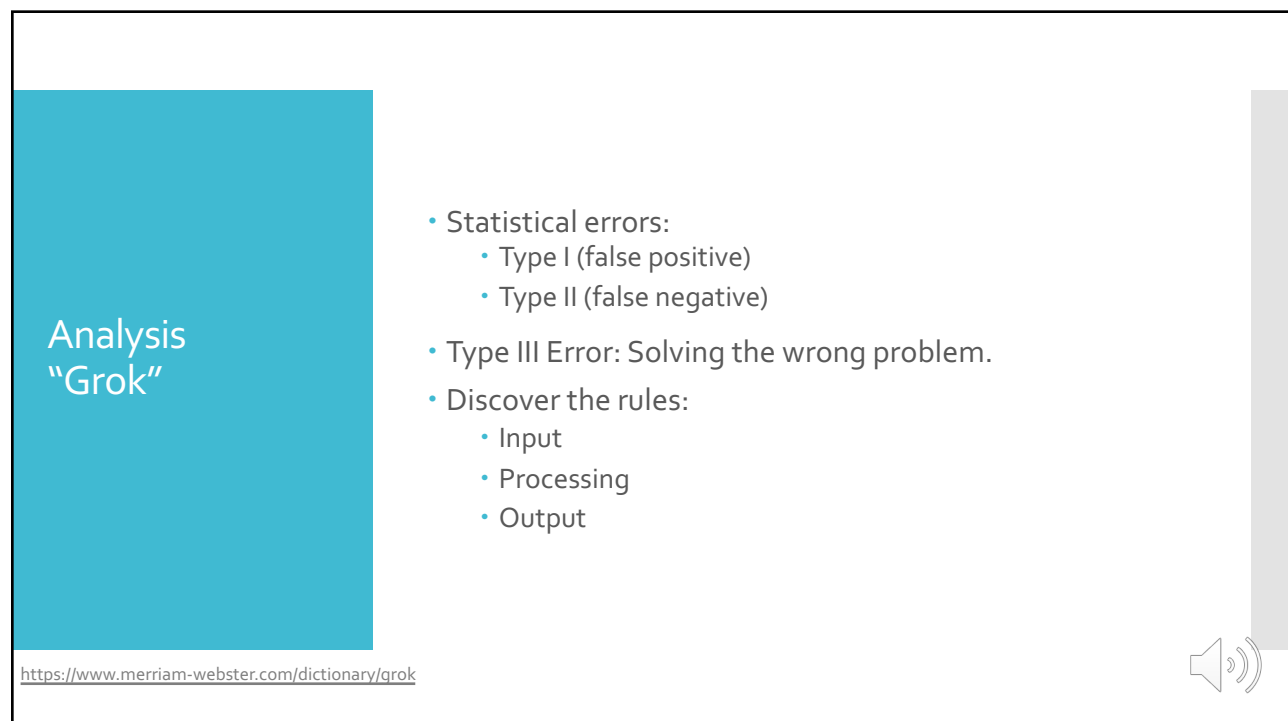
"Writing a Program"

Analysis

- Requirements
- Inputs
- Outputs
- Processing

Speaker icon

5



Analysis  
"Grok"

- Statistical errors:
  - Type I (false positive)
  - Type II (false negative)
- Type III Error: Solving the wrong problem.
- Discover the rules:
  - Input
  - Processing
  - Output

<https://www.merriam-webster.com/dictionary/grok>

Speaker icon

6

## Design

- You are designing a solution
  - Forethought
  - Competing ideas ► choices
- Data structures
- Algorithms
- Partitioning of functionality
  - Tight cohesion; loose coupling
- Can result in an outline of the program
  - Could read like the outline of an essay
  - Could even write the function declarations



7

## Code, Test

### Two Extremes

1. Write everything and see what happens
  - Code, code, code
  - Test
2. Write atomic units of code, test and incorporate into the whole.
  - Code, test
  - Code, test
  - Code, test



8

## Consider the Assignment

- How many wrote it all and then started testing?
- How else could you have done it?
- How could you develop such that the code you have always works (but may be incomplete)?



9

## Process, Process, Process

- Success is a function of process
  - No process ► low chance of success
  - Bad process ► low chance of success
  - Good process ► high chance of success
- What do you think?
- What does a good process look like?
  - Automation wherever possible
  - Tight edit/compile loop
  - Testing: correctness



10



# make

Think ahead. Think less. Compile less.



11



## Building a Multi-File Program

- A program made of 2 files:
  - `areDistinct.c`
  - `main.c`
- Two processes
  - Compile
  - Link



12

## Linking

- Linker combines object files produced by compiler, along with library object files, to produce executable file
- Linker has to resolve external references left behind by the compiler
  - function defined in one file and called in another file
  - function has to be declared before it can be called, but doesn't need to be defined in same file



13

## One-Step Build

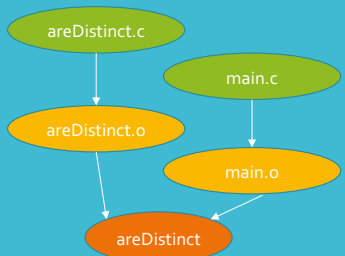
```
gcc -o areDistinct areDistinct.c main.c
```

- Two object files produced by compiler automatically passed to linker



14

## Two-Step Build



```
gcc -c areDistinct.c
```

```
gcc -c main.c
```

```
gcc -o areDistinct areDistinct.o main.o
```

- Which commands must be re-executed if a change is made to `areDistinct.c`?
- We say that `areDistinct.o` depends on `areDistinct.c`
- Also, `areDistinct` depends on `areDistinct.o` and `main.o`
- Dependencies also occur via header files (to be seen later)



15

## Possible Solutions

- Keep everything in one file
  - editing is slow
  - compiling is slow
- Keep things in separate files, but always recompile everything
  - compiling is slow
- Automatically chose things to be recompiled
  - use make utility



16



## coNsISTeNT Compile/Link Process

- Make: a command line utility
- Uses a makefile specifying targets and how to achieve them
- Especially useful for multi-file programs



17

## Makefile (I)

- Using an editor, create a file named `makefile` or `Makefile`
- This file specifies two things:
  - the dependencies
  - the shell commands necessary to make a new version of a file
- The idea is that by simply typing "make", (only) the appropriate commands will execute



18

## Makefile (II)

```
areDistinct: areDistinct.o main.o
    gcc -o areDistinct areDistinct.o main.o
areDistinct.o: areDistinct.c
    gcc -c areDistinct.c
main.o: main.c
    gcc -c main.c
```

- *Note that tab character must be used before each executable line*



19

## Making a Target

- You can specify several targets in a single makefile
  - By default the first target is the only one examined (along with any rules for its dependencies) Can build any target in the description file:
- ```
$ make target
```
- If no prerequisite files modified since last time target was created, make issues the message:
- ```
'target' is up to date
```



20

## Using make for Testing

```
test0: areDistinct
    ./areDistinct inputs/input0 > outputs/output0
    diff expected/expected0 outputs/output0
```

- To test:

```
$ make test0
```

- Then add other tests



21

## Cleaning up with make

```
clean:
    /bin/rm -f *.o areDistinct outputs/*
```



22