ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Informatik I (D-ITET)

Makefiles (A short introduction) 28/11/2012

UPN (Reverse Polish Notation) Calculator

- **Suppose you want to implement a UPN Calculator for complex numbers, e.g.**

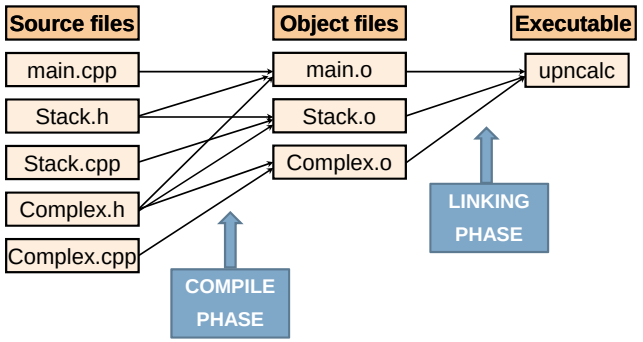
```
./upncalc 12.2+i3.2 14.0+i5.9 +
Returns the result of : (12.2+i3.2) + (14.0+i5.9) = 26.2 + i8.1
```

or

```
./upncalc 12.2+i3.2 14.0+i5.9 x
Returns the result of : (12.2+i3.2) x (14.0+i5.9) = 151.92 + i116.78
```

- **Also suppose you want to implement everything with classes and stacks**
- **Therefore you have the following files:**
  - **`Complex.h , Complex.cpp`** for the class that implements complex numbers arithmetics.
  - **`Stack.h , Stack.cpp`** for the class that implements the stack representation.
  - **`main.cpp`** for the source file that contains the main function, input from the user etc.

Compiling large projects

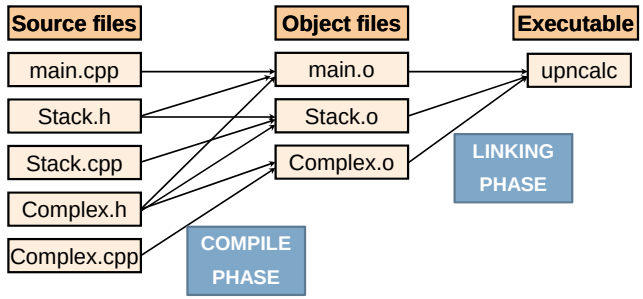


In one step:

```
g++ -Wall -pedantic -ansi Complex.cpp Stack.cpp main.cpp  -o upncalc
```

- When one file changes, ALL have to be compiled again.

Compiling large projects



Compiling and Linking separately

Compile source files but not link

```
g++ -Wall -pedantic -ansi -c Complex.cpp
g++ -Wall -pedantic -ansi -c Stack.cpp
g++ -Wall -pedantic -ansi -c main.cpp
g++ -Wall -o upncalc Complex.o Stack.o main.o
```

- "Expensive" to list everything.

What is a makefile?

- **Makefiles** are files consisting of rules and dependencies that specify how to build (or "make") a target program.
- **Make** is a utility that automatically builds executable programs from source code by reading makefiles.

- The dependencies are needed in order to determine when the target needs to be reconstructed (usually by recompiling the code).
- The **make** utility decides when to reconstruct the target based on the **timestamps** of the dependency files**.**
- If the dependency files have a more recent timestamp than the target, then the target has to be rebuilt.
- The updates are done recursively. When checking the dependencies, the **make** utility checks if the dependencies are also targets and if so, it looks at their dependencies and so on.

Makefiles

- **Structure of a statement in a Makefile**

```
TARGET: DEPENDENCY1, DEPENDENCY2, …
<TAB>    TARGET_COMMAND1    <ENTER>
<TAB>    TARGET_COMMAND2    <ENTER>
 ……
```

For example:

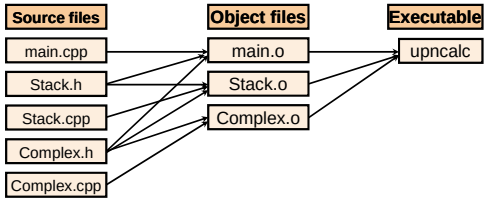
```
upncalc: main.o Stack.o Complex.o
    g++ -Wall -o upncalc main.o Stack.o Complex.o
```

This is translated as:

" Target *upncalc* depends on the object-files *main.o, Stack.o* and *Complex.o* and in order to create the target we need to call the g++ command listed"

- You should name your makefile as (in order of preference) :
  - makefile (make utility automatically searches for it when you type: `make`)
  - Makefile (make utility automatically searches for it when you type: `make`)
  - this_is_my_makefile (you have to type: `make –f this_is_my_makefile`)

Makefile **Version 0**

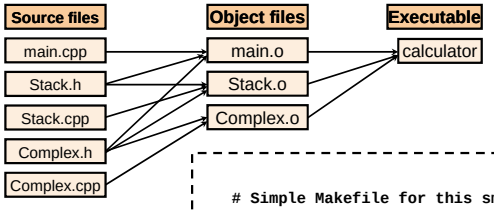



```
# Simple Makefile for this small project

upncalc: main.o Stack.o Complex.o
    g++ -Wall Complex.cpp Stack.cpp main.cpp -o upncalc
```

But… Does this make any sense?

Makefile **Version 1**




```
# Simple Makefile for this small project

calculator: main.o Stack.o Complex.o
    g++ -Wall -o calculator main.o Stack.o Complex.o

main.o: main.cpp Stack.h Complex.h
    g++ -Wall -c main.cpp

Stack.o: Stack.cpp Stack.h Complex.h
    g++ -Wall -c Stack.cpp

Complex.o: Complex.cpp Complex.h
    g++ -Wall -c Complex.cpp
```

!! Here you need a TAB

!! No TAB on empty lines

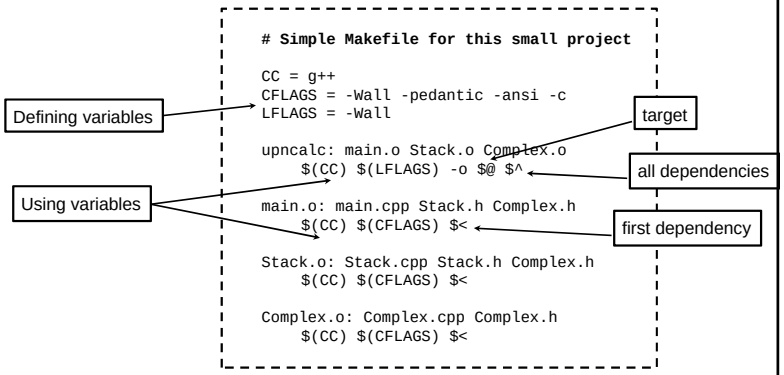
Makefiles : Variables

- **Defining the variables:** `VARNAME = value`
- **Accessing the variables:** `$(VARNAME)`
- **Some predefined variables**:
  - `$@` : Filename of the target (synonym `$(target)` in makepp)
  - `$<` : Name of the first dependency (synonyms : `$(dependency)` in makepp)
  - `$^` : Names of all dependencies (synonyms : `$(dependencies)` in makepp)

- **Typical Variables defined:**
  - `OBJS = main.o Stack.o Complex.o`
  - `CC = g++`
  - `DEBUG = -g`
  - `CFLAGS = -Wall -c $(DEBUG)`
  - `LFLAGS = -Wall $(DEBUG)`

Makefile **Version 2**

- **The Makefile from Version 1 using variables:**



Makefiles : Wildcards

- **It's a bit complicated to define a separate target for each source file.**
- **With pattern rules one can combine similar targets.**
- **Example:**

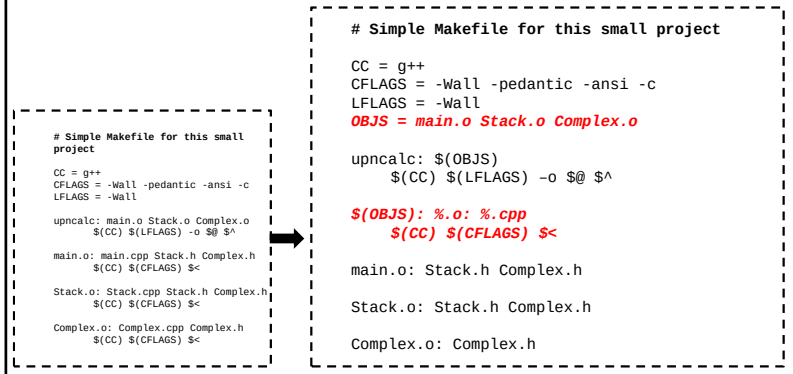
```
$(OBJS): %.o: %.cpp
```

➡ **<u>This means:</u> All object files (.o) with the same name as the source files (.cpp)**

⚡ **<u>Note that:</u> Dependencies for the header files (.h) are not taken into account here.**

Makefile **Version 3**

- **The Makefile using variables and wildcards:**



Makefiles : Phony (Dummy) Targets

- **A phony target** is a target that is not really the name of a file
- Rather it is **just a name for a recipe to be executed** when you make an explicit request.
- A **phony target should not be a prerequisite of a real target**. If it is, its recipe will be run every time **make** goes to update that target.
- There are two reasons to use a phony target:
  - **to avoid a conflict with a file of the same name, and**
  - **to improve performance.**
- **Example:**

```
.PHONY: clean
clean:
    rm -f *.o calculator
```

- It is executed only if the target is called explicitly and regardless if a file named clean is present.
- To execute the phony target: **`make clean`**

Makefile **Version 4**

- **The Makefile using variables, wildcards and *phony target*:**

```
CC = g++
CFLAGS = -Wall -pedantic -ansi -c
LFLAGS = -Wall
OBJS = main.o Stack.o Complex.o

upncalc: $(OBJS)
	$(CC) $(LFLAGS) -o $@ $^

$(OBJS): %.o: %.cpp
	$(CC) $(CFLAGS) $<

main.o: Stack.h Complex.h

Stack.o: Stack.h Complex.h

Complex.o: Complex.h
```

```
# Simple Makefile for this small project

.PHONY: clean
CC = g++
CFLAGS = -Wall -pedantic -ansi -c
LFLAGS = -Wall
OBJS = main.o Stack.o Complex.o

upncalc: $(OBJS)
    $(CC) $(LFLAGS) -o $@ $^

$(OBJS): %.o: %.cpp
    $(CC) $(CFLAGS) $<

main.o: Stack.h Complex.h

Stack.o: Stack.h Complex.h

Complex.o: Complex.h

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


Makefiles : Dependencies by g++

- **g++ can deal with all local dependencies in make-format**

```
g++ -MM <cpp-files>
```

- **One can integrate into the Makefile**

```
depend:
    g++ -MM $(SRCS) > deps.mk
-include deps.mk
```


Makefile **Version 5**

- **The Makefile using variables, wildcards, *phony target and g++ dependencies*:**

```
.PHONY: clean
CC = g++
CFLAGS = -Wall -pedantic -ansi -c
LFLAGS = -Wall
OBJS = main.o Stack.o Complex.o

upncalc: $(OBJS)
	$(CC) $(LFLAGS) -o $@ $^

$(OBJS): %.o: %.cpp
	$(CC) $(CFLAGS) $<

main.o: Stack.h Complex.h

Stack.o: Stack.h Complex.h

Complex.o: Complex.h

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

```
# Simple Makefile for this small project

.PHONY: clean depend
CC = g++
CFLAGS = -Wall -pedantic -ansi -c
LFLAGS = -Wall
OBJS = main.o Stack.o Complex.o
SRCS = main.cpp Stack.cpp Complex.cpp

upncalc: $(OBJS)
    $(CC) $(LFLAGS) -o $@ $^

$(OBJS): %.o: %.cpp
    $(CC) $(CFLAGS) $<

clean:
    rm -rf *.o upncalc deps.mk

depend:
    $(CC) -MM $(SRCS) > deps.mk

-include deps.mk
```

Common errors in makefiles

- Don't put a TAB at the beginning of commands.
  This will result in the command not running.
- Put a TAB at the beginning of blank lines.
  Then the **make** utility will complain for a "blank" command.
- Not getting the dependencies correct.

QUESTIONS ??

For more info:

`http://www.gnu.org/software/make/manual/`

`http://makepp.sourceforge.net/1.19/makepp_tutorial.html`