# Working with multiple files

Stefan D. Bruda

CS 464/564, Fall 2023

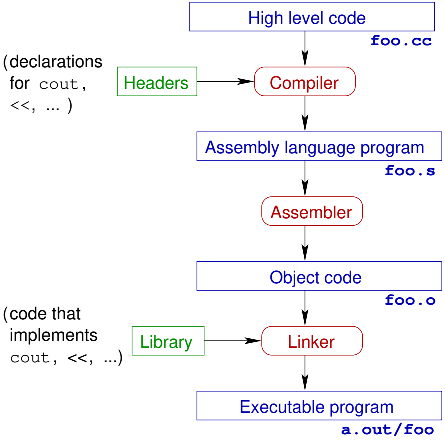
# Splitting the code

- Often we want to split our program into multiple files (or modules)
- Advantages: encapsulation, reusability, size
    - Compilation time also reduced
- A module consists of two parts:
    - the header file, where all the declarations available outside the module go (e.g., `tcp-util.h`)
    - the C/C++ code which implements the things declared in the header (e.g., `tcp-util.cc`)
- Another module (say `main.cc`) that wants to use `tcp-util.cc` will use the directive

  ```
  #include "tcp-util.h"
  ```

    - Then `tcp-util.cc` and `main.cc` will be compiled and linked together
    - We can automate this process by encoding the recipe into a makefile

# Bringing your program to life



High level code

foo.cc

(declarations for cout, <<, ... )

Headers

Compiler

Assembly language program

foo.s

Assembler

Object code

foo.o

(code that implements cout, <<, ...)

Library

Linker

Executable program

a.out/foo

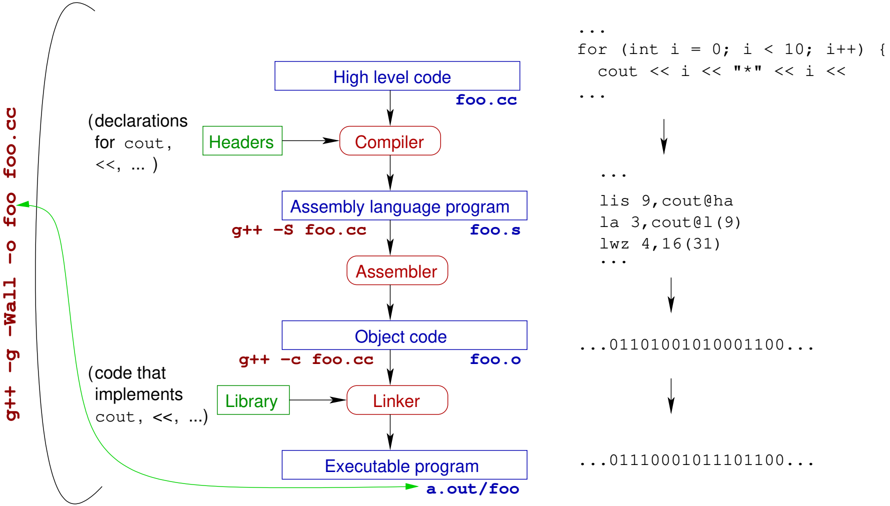
```
...
for (int i = 0; i < 10; i++) {
  cout << i << "*" << i <<
...
```

```
...
lis 9,cout@ha
la 3,cout@l(9)
lwz 4,16(31)
...
```

```
...0110100101000110O...
```

```
...0111000101110110O...
```

# BRINGING YOUR PROGRAM TO LIFE (CONT'D)



g++ -g -Wall -o foo foo.cc

High level code

foo.cc

(declarations for cout, <<, ... )

Headers

Compiler

Assembly language program

g++ -S foo.cc

foo.s

Assembler

Object code

g++ -c foo.cc

foo.o

(code that implements cout, <<, ...)

Library

Linker

Executable program

a.out/foo

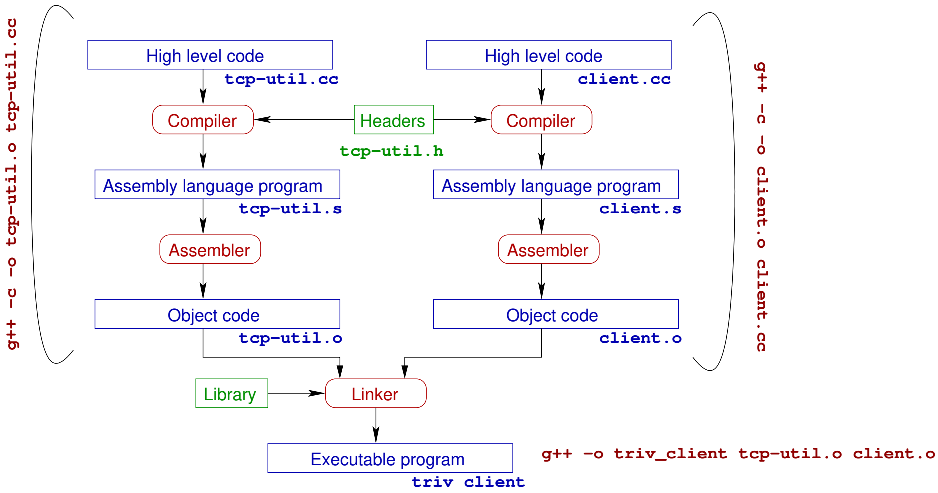
```
...
for (int i = 0; i < 10; i++) {
  cout << i << "*" << i <<
...
```

```
...
lis 9,cout@ha
la 3,cout@l(9)
lwz 4,16(31)
...
```

```
...01101001010001100...
```

```
...01110001011101100...
```

# PUTTING MANY MODULES TOGETHER



```
g++ -c -o tcp-util.o tcp-util.cc
```

High level code

tcp-util.cc

Compiler

Headers

tcp-util.h

Assembly language program

tcp-util.s

Assembler

Object code

tcp-util.o

High level code

client.cc

Compiler

Assembly language program

client.s

Assembler

Object code

client.o

```
g++ -c -o client.o client.cc
```

Library

Linker

Executable program

triv client

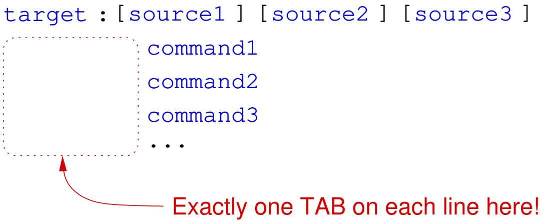
```
g++ -o triv_client tcp-util.o client.o
```

# Makefiles

- A makefile contains recipes for compiling multiple file programs
- A makefile contains macrodefinitions, e.g.,

```
# this is a comment
CXX = g++
CXXFLAGS = -g -Wall
```

- Then we have rules of the form:



```
target : [source1 ] [source2 ] [source3 ]
	command1
	command2
	command3
	. . .
```

Exactly one TAB on each line here!

  - A target is the name of the file to be produced
    - It is produced by executing the corresponding commands
  - The sources are the files needed to produce the target (if any)
    - They form a dependency tree

# MAKEFILES (CONT'D)

- Sample makefile:

```
all: triv_client

tcp-utils.o: tcp-utils.h tcp-utils.cc
        $(CXX) $(CXXFLAGS) -c -o tcp-utils.o tcp-utils.cc

client.o: tcp-utils.h client.cc
        $(CXX) $(CXXFLAGS) -c -o client.o client.cc

triv_client: client.o tcp-utils.o
        $(CXX) $(CXXFLAGS) -o triv_client client.o tcp-utils.o

clean:
        rm -f triv_client *~ *.o *.bak core \#*
```

- Suppose you type make target in some directory d
    - `make` without arguments produces the first target in the makefile
- The command looks for a file called `Makefile` in d and follows all the necessary rules therein along the dependency tree to produce the file target
- All the targets needed by target (based on said dependency tree) are also made, unless they are up to date (decision based on modification times)