Lecture 14: Writing Large Programs

Sarah Nadi
nadi@ualberta.ca
Department of Computing Science
University of Alberta

CMPUT 201 - Practical Programming Methodology

[With material/slides from Guohui Lin, Davood Rafei, and Michael Buro. Most examples taken from K.N. King's book]

Agenda

- Dividing a program into files
- Source files
- Header files
- Building a multiple-file program

Readings

- Textbook Chapter 15

A C Program

- Even though we have mostly been writing single C-file programs, a C program can consist of multiple files
- There are mainly two types of files in a C program:
  - *source files* which contain the definitions (i.e., the implementation/code) of functions and variables
  - *header files* which contain directives and function prototypes to be shared among source files

Example: Reverse Polish Notation (RPN)

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

- Suppose we want to write a program that evaluates an RPN expression

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

- Suppose we want to write a program that evaluates an RPN expression

- We can easily do this if we have a stack:

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

- Suppose we want to write a program that evaluates an RPN expression

- We can easily do this if we have a stack:

  - Read a token

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

- Suppose we want to write a program that evaluates an RPN expression

- We can easily do this if we have a stack:

  - Read a token

  - If the token is a number, push it onto the stack

Example: Reverse Polish Notation (RPN)

- The reverse Polish notation has operators follow the intended operands. For example, 30 5 - 7 * is actually (30 - 5) * 7

- Suppose we want to write a program that evaluates an RPN expression

- We can easily do this if we have a stack:

  - Read a token

  - If the token is a number, push it onto the stack

  - If the token is an operator, pop the top the last two numbers from the stack, perform the operation, and push their results on to the stack

Functionality in the RPN Calculator Program

pop token

read token from input

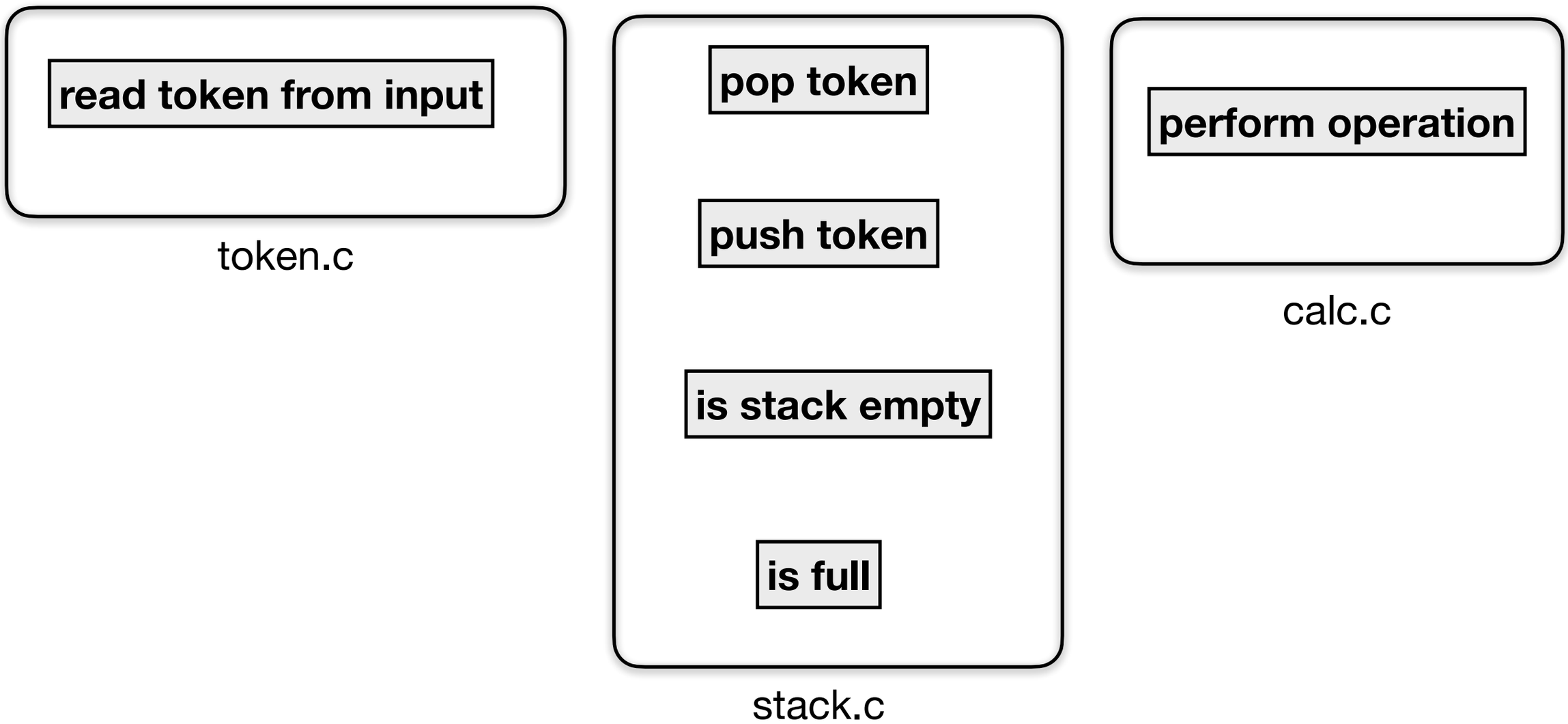
is stack empty

push token

perform operation

is full

Functionality in the RPN Calculator Program



Splitting a Program into Multiple Files

- Grouping related functions and variables into a single file helps clarify the structure of the program

- Each source file can be compiled separately, which saves a lot of time if the program is large and certain parts are frequently changed (will revisit this when discussing multi-source Makefiles in a couple of slides)

- Functions are more easily reused in other programs when grouped in separate source files. In our example, we can reuse all the token functionality without the stack functionality or vice versa.

Header Files

- To allow a function in one file to call a function in another file or access an external variable in another file, we need to share information across files

- Creating a header file and using the `#include` directive allows us to do so. Remember that the #include directive tells the preprocessor to open the specified files and include its contents in the current file

- `#include <filename>` — refers to header files that belong to the C library

- `#include "filename"` — refers to all other header files

Sharing Macro Definitions and Type Definitions

```
typedef int Age;
#define DRIVING_AGE 16
#define LEGAL_AGE 18
```

age.h

```
#include "age.h"

int check_driving_record(Age age){
   int time = age - DRIVING_AGE;
}
```

driving_record.c

```
#include "age.h"

int check_criminal_record(Age age){
   if (age >= LEGAL_AGE)
      //treat as adult
   else
      //treat as juvenile
}
```

legal_check.c

Sharing Function Prototypes

```
int main(){

  …
  int token = …;
  push(token);
  …
}
```

calc.c

Sharing Function Prototypes

```
int main(){

  …
  int token = …;
  push(token);
  …
}
```

calc.c

What is “push” ? The compiler does not know what this function is so it has no way of checking if it has been used correctly or not

Sharing Function Prototypes — Solution 1

```
void push(int t);

int main(){

  …
  int token = …;
  push(token);
  …
}
```

calc.c

Sharing Function Prototypes — Solution 1

```
void push(int t);

int main(){

  …
  int token = …;
  push(token);
  …
}
```

calc.c

This solves the problem of telling the compiler what the function is. However, it isn't a good solution from a reuse/maintenance perspective, because what if this function is used in other files. Should each file then write its own prototypes? What if these prototypes are inconsistent? or What if we decide to change the function

Sharing Function Prototypes — Solution 2 (the better one)

```
void make_empty();
int is_empty();
int is_full();
void push (int i);
int pop(void);
```

stack.h

```
#include “stack.h"

int main(){

  …
  int token = …;
  push(token);
  …
}
```

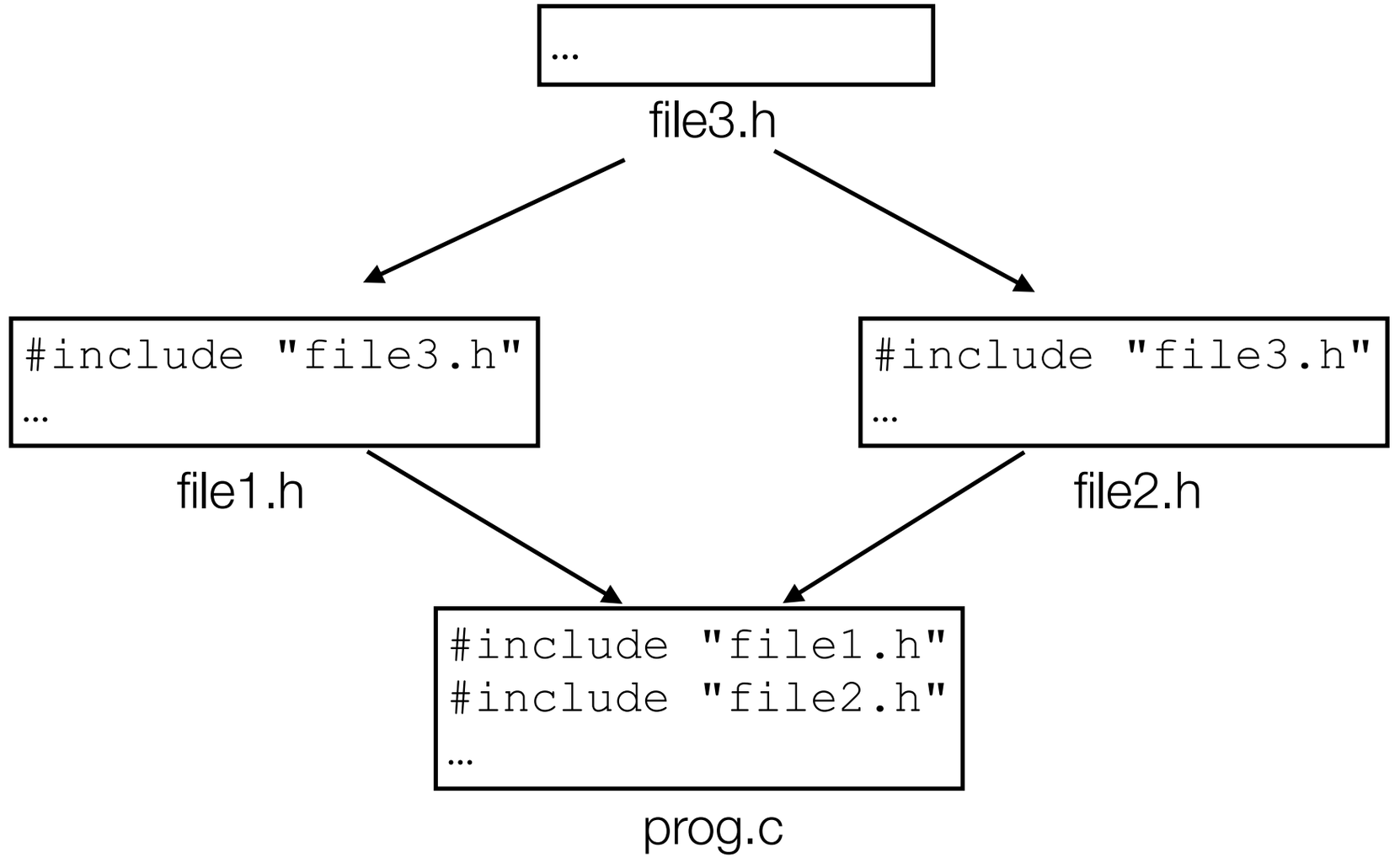
calc.c

```
#include "stack.h"

int contents[100];
int top = 0;

void make_empty(){ … }
int is_empty(){ … }
int is_full() { … }
void push(int i){ … }
int pop(){ … }
```

stack.c

Problem of Including Same Header File Twice

- If a source file includes the same header file twice, compilation errors may occur. The error happens when the compiler sees, for example, the same type defined again.



Solving the Problem: Protecting Header Files

```
#ifndef FILE_3_H
#define FILE_3_H
typedef int Age;
#endif
```

demo: doubleinclude/

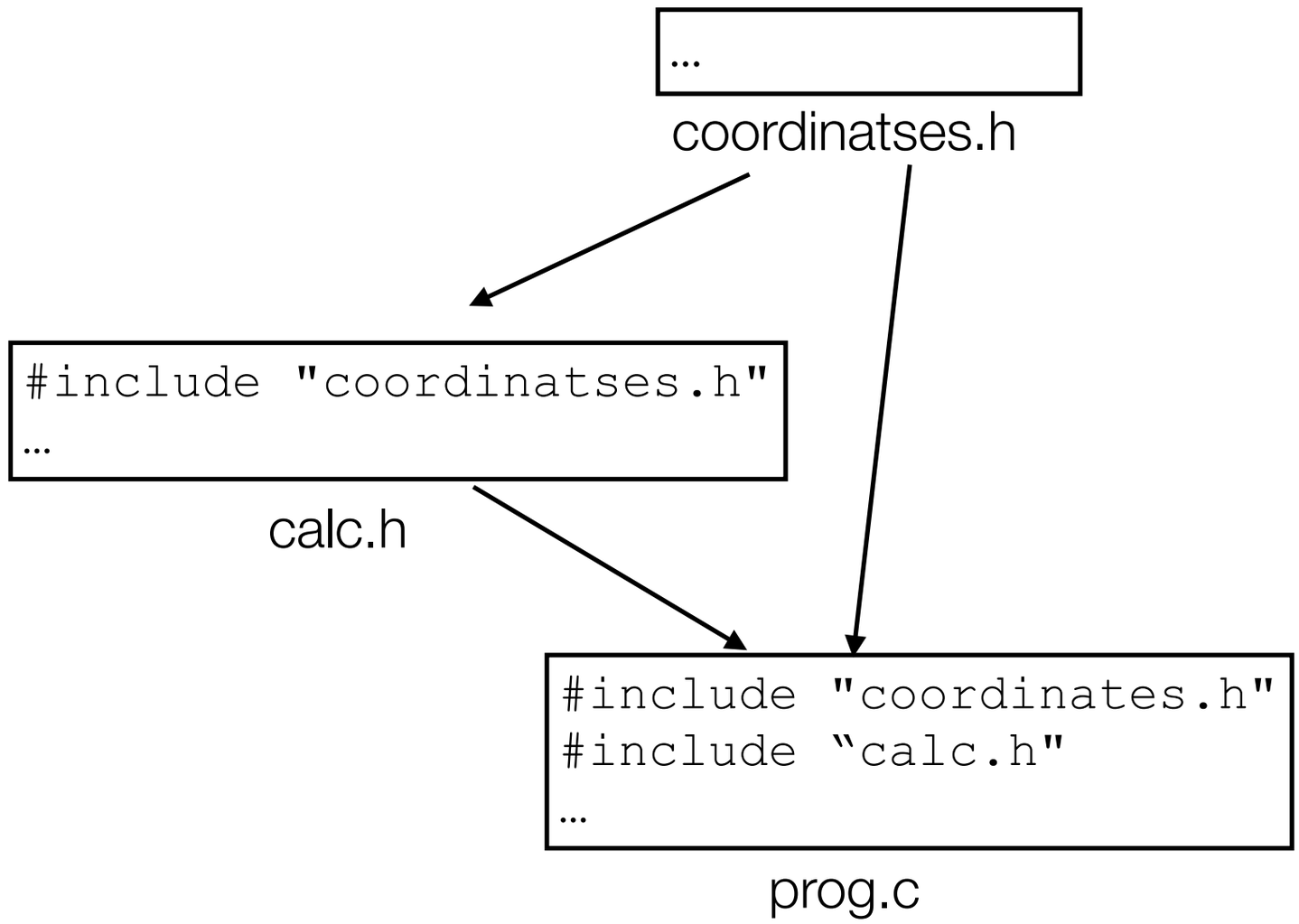
Solving the Problem: Protecting Header Files

```
#ifndef FILE_3_H
#define FILE_3_H
typedef int Age;
#endif
```

The first time the preprocessor goes to include the file, FILE_3_H will not be defined, so the preprocessor will define it and proceed to copy the contents of the file. The second time the preprocessor goes to include the file, FILE_3_H will be defined so the #ifndef will be false, and no contents will be copied. This prevents including the same content twice.

demo: doubleinclude/

demo: doubleinclude/



Building a Multiple-file Program

- Building a large program requires the following steps:

  - **Compiling**: each source file in the program must be compiled separately. For each source file, the compiler generates a file containing object code. These files, known as *object files* have the extension .o in UNIX and .obj in Windows.

  - **Linking**: The linker combines the .o files created in the previous step — along with code for library functions — to produce an executable file. The linker is responsible for resolving external references left behind by the compiler. An external reference occurs when a function in one file calls a function defined in another file or accesses a variable defined in another file.

Building in a Single Step

```
gcc -Wall -std=c99 -o calc calc.c stack.c token.c
```

Building in a Single Step

```
gcc -Wall -std=c99 -o calc calc.c stack.c token.c
```

It's tedious to always write this out and we could waste a lot of time recompiling code from all files, and not just the ones affected by the most recent changes

Makefiles

- Makefiles make it easier to build large programs and supports rebuilding only files affected by recent changes

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

Makefiles

- Makefiles make it easier to build large programs and supports rebuilding only files affected by recent changes

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

-c option tells the compiler to preprocess, compile, and assemble the .c file but not attempt to link it. Remember that linking at this stage may fail since not all definitions of all functions have been seen yet. By default, the output of -c is put into a file with the same name but with .o extension instead of .c.

Makefiles

called *targets* can run:
`make target`
If no target is specified, the first one executes by default

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

Makefiles

called *targets*
can run:
`make target`
If no target is specified, the first one executes by default

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

called *dependencies.*
A modification in one of the dependencies causes the target to be rebuilt

Makefiles

called *targets*
can run:
`make target`
If no target is specified, the first one executes by default

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

called *dependencies.*
A modification in one of the dependencies causes the target to be rebuilt

called the *command..* this is the command that gets executed if the target is (re)built.. a.k.a *recipe*

Makefiles

called *targets*
can run:
`make target`
If no target is specified, the first one executes by default

```
calc: calc.o stack.o tokens.o
  gcc -Wall -std=c99 -o calc calc.o stack.o tokens.o

calc.o: calc.c stack.h tokens.h
  gcc -Wall -std=c99 -c calc.c

stack.o: stack.c stack.h
  gcc -Wall -std=c99 -c stack.c

tokens.o: tokens.h tokens.c
  gcc -Wall -std=c99 -c tokens.c

clean:
  rm -f *.o
  rm -f calc
```

called *dependencies.*
A modification in one of the dependencies causes the target to be rebuilt

The whole group of target, dependencies, and command is called a *rule*

called the *command..* this is the command that gets executed if the target is (re)built.. a.k.a *recipe*

demo: check-record/

Notes about Makefiles

- If you just type `make`, it will build the first target
- You can also explicitly build a target by saying `make targetname`. E.g. from previous slide: `make clean` or `make token.o`