C Program Structure

CSC 230 : C and Software Tools

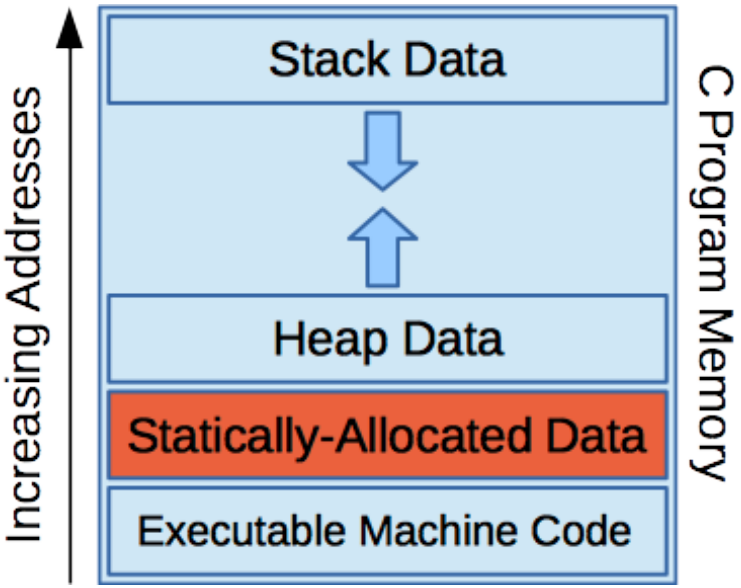
NC State Department of Computer Science

Topics for Today

- Memory and Storage Class
- Multi-file programs
  - Global variable and function declaration
  - Header files
- Compile and link
- Build automation with Make

Storage Class

- A variable's *storage class* determines
  - Its lifetime
  - Where it's allocated
  - How it's initialized
- Global variables have *static storage* class.



Static/Global Variable Initialization

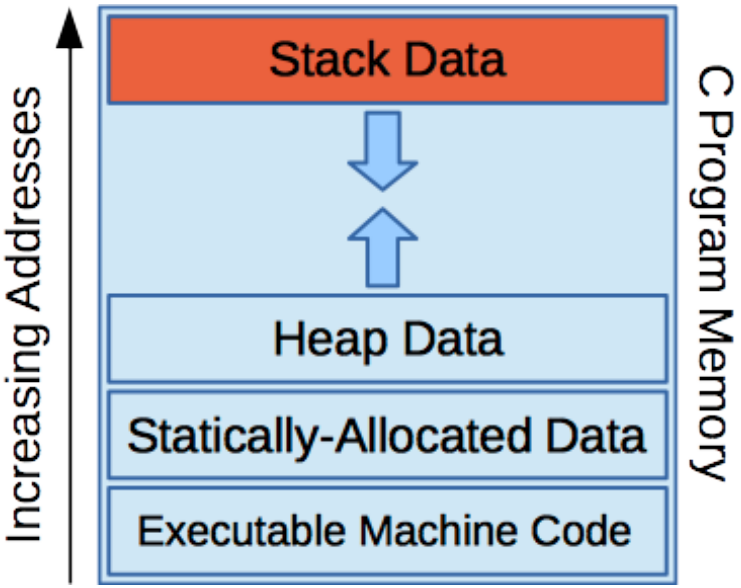
- Variables with static allocation (e.g., globals) are initialized at program start-up.
- If you don't initialize them, they automatically get zero.
- If you do initialize them, you need a constant expression

```
char globalC = 'a';                 // Yes
int globalI = 15 + (39 % 3);        // Yes
int globalJ = globalI + 1;          // Nope
```

Storage Class

- Storage for local variables is automatically allocated
  - When control enters their containing block
    - Maybe when the function starts, it's the compiler's choice.
  - These variables have the *auto* storage class
    - We'll also call them *local* or *stack* variables.
- Auto is even a keyword:

```
auto int i = 5;
```



Auto Variable Initialization

- If you don't initialize them … you get whatever's left over in that part of stack memory.

```
for ( int i = 0; i < 100; i++ ) {
    int a;
    ...;
}
```

Who knows what value this will have. A common mistake.

- If you do initialize … the initialization is evaluated each time control passes the declaration.

```
for ( int i = 0; i < 100; i++ ) {
    int a = pow( 3.25, i ) + sqrt( i + 1 );
    ...;
}
```

This gets evaluated on every iteration.

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

Memory

x: 10

Execution trace

```
  cooking();
```

Variable Allocation/Initialization

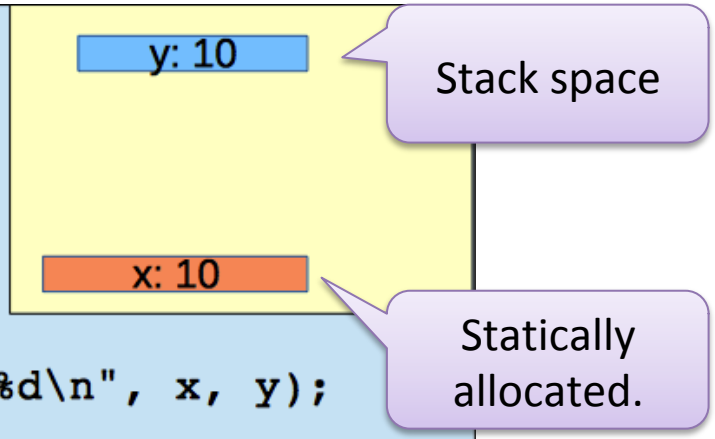
```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

x: 10

```
cooking();
```

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```




```
  cooking();
```

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 9

x: 11

```
  cooking();
```

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 9

z: 9

x: 11

```
cooking();
```

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 9

z: 8

x: 12

```
  cooking();
```

Variable Allocation/Initialization

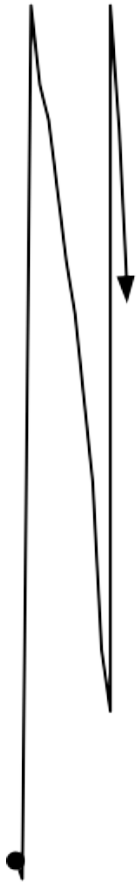
```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 9
z: 8
y: 12

x: 12

```
  cooking();
```

Variable Allocation/Initialization



```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

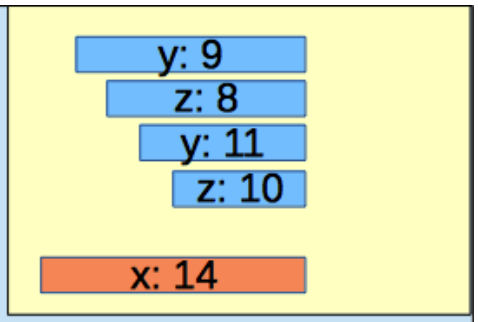
y: 9
z: 8
y: 11

x: 13

```
  cooking();
```

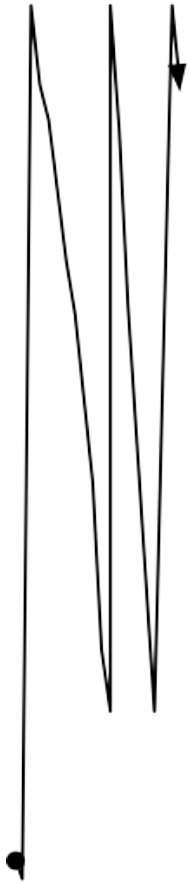
Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```




```
  cooking();
```

Variable Allocation/Initialization



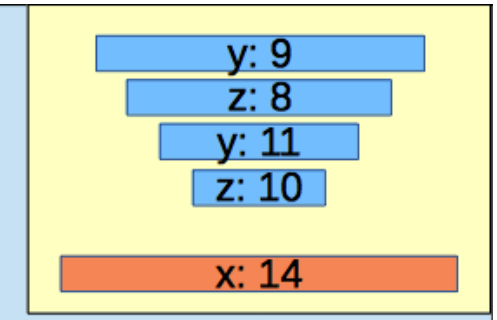
```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 9
z: 8
y: 11
z: 10
x: 14

```
  cooking();
```

Variable Allocation/Initialization

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    int y = x;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```



```
  cooking();
```

Static Local Variables

- Scope isn't the same as storage class.
  - You can have local variables that are statically allocated
  - How? Using the keyword `static`.

```
for ( int i = 0; i < 100; i++ ) {
    static int a = 10 + EXTRA;
    ...;
}
```

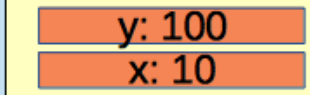
I just get created and initialized once.

- It's like a variable that has the lifetime of a global, but the scope of a stack variable.
- Like a global, requires a constant initialization expression.

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

```
  cooking();
```



A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 100
x: 10

```
cooking();
```

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

y: 99
x: 11

```
  cooking();
```

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

z: 99

y: 99

x: 11

```
  cooking();
```

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

z: 98

y: 99

x: 12

```
  cooking();
```

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

z: 98

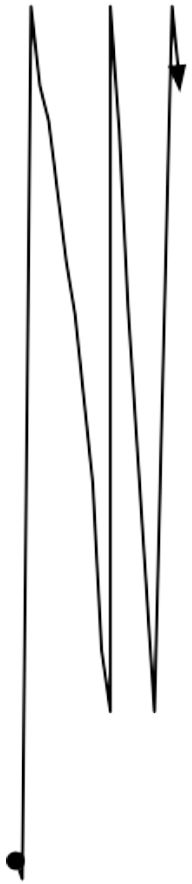
z: 98

y: 98

x: 13

```
cooking();
```

A Static Local Variable



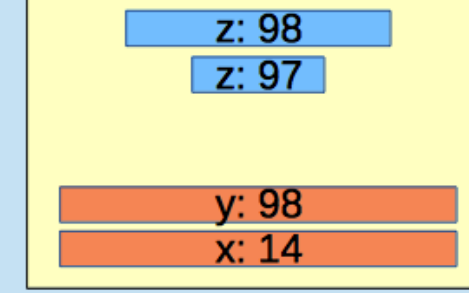
```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```

z: 98
z: 97
y: 98
x: 14

```
  cooking();
```

A Static Local Variable

```
int x = 10;

void cooking()
{
  if ( x < 14 ) {
    static int y = 100;
    x++;
    y--;
    printf("x = %d, y = %d\n", x, y);
    cleaning( y );
  }
}

void cleaning( int z )
{
  x++;
  z--;
  printf("x = %d, z = %d\n", x, z);
  cooking();
}
```



```
  cooking();
```

Variable Initialization Summary

- Local variables
  - Initialized by re-evaluating their initialization expression each time they are created.
  - Or, not initialized at all if there's no initialization expression.
- Global variables
  - Initialized once at the start of execution via their constant initialization expression
  - Or initialized to zero if you don't supply an expression.
- Static local variables
  - Just like global

The `register` Storage Class

- Local variables don't have to be stored on the stack (although they usually are).
- We can make a recommendation that the compiler use a CPU register

```
int total = 0;
for ( int i = 0; i < 1000000000; i++ )
  total += i;
```

Allocated on the stack.

Maybe not.

```
int total = 0;
for ( register int i = 0; i < 1000000000; i++ )
  total += i;
```

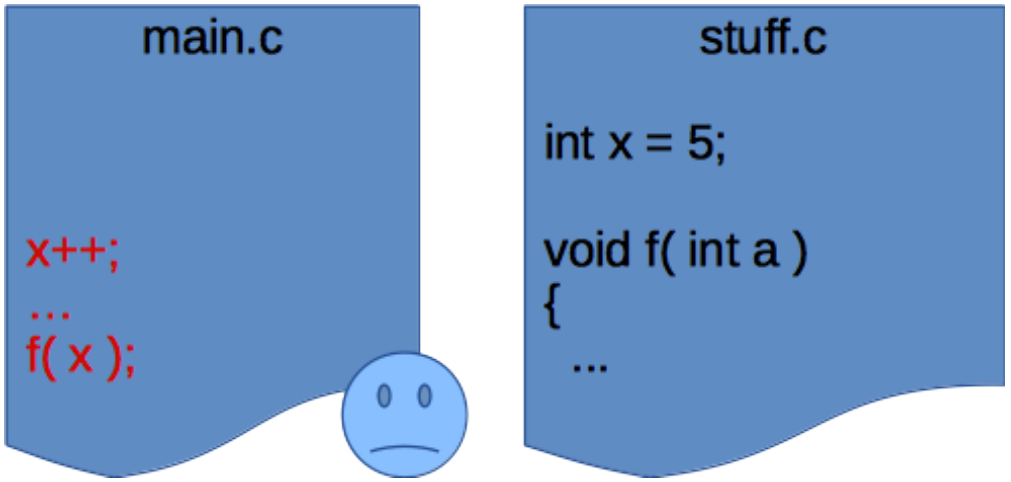
Consider storing i in a CPU register.

Making Suggestions

- `register` can only be specified for **`auto`** variables (i.e., not for global variables)
  - There are other restrictions on what types of variables can be specified as **`register`**
  - … and on what you can do with these variables
- Less important than it once was.
  - Optimizing compilers may be able to do a better job than you at allocating scarce CPU registers

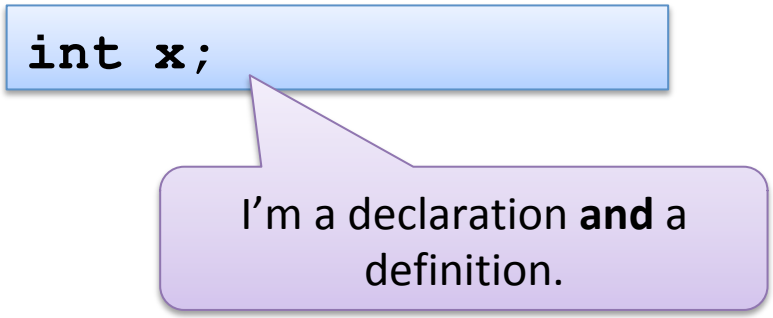
Accessing External Symbols

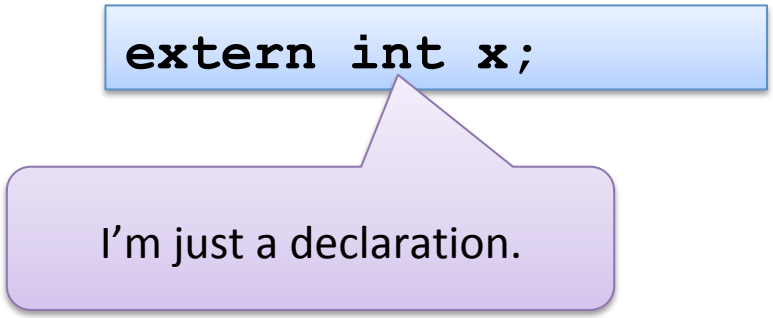
- We can make **functions** and **global variables** that are **visible to multiple source files**.
- But, the compiler looks at each source file as an individual *compilation unit.*
  - it won't let us use symbols it has never heard of.



Extern Variables

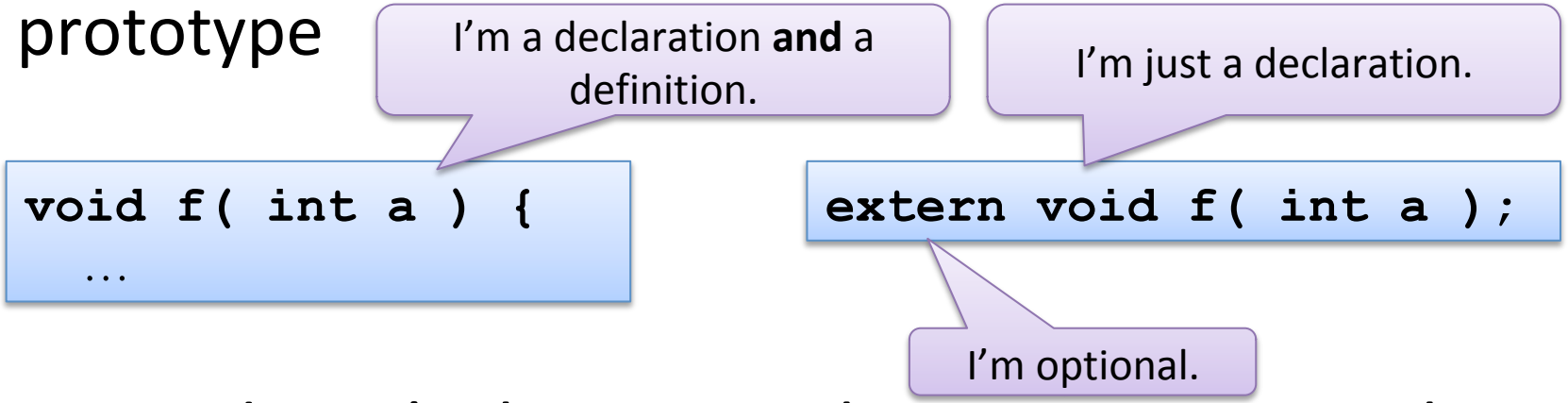
- That's what the `extern` keyword is for
  - It tells the compiler about a something that's defined elsewhere.
  - It lets us describe a variable without allocating space for it.
  - So, it lets us declare a variable without defining it.





Extern and Prototypes

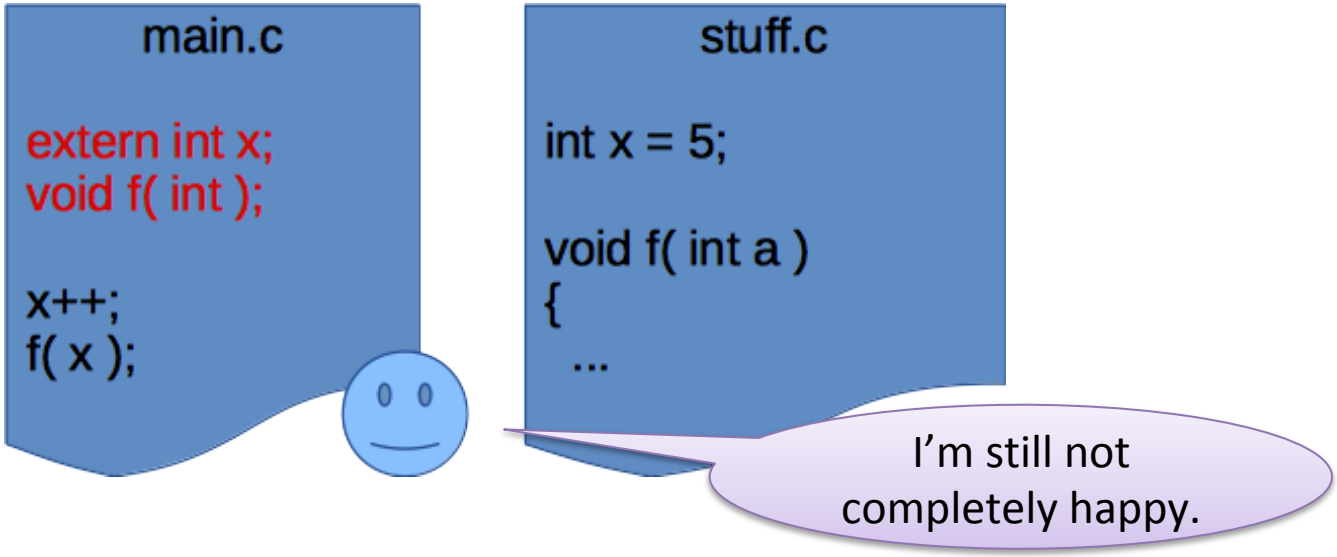
- This is the same kind of thing a prototype does.
  - It tells the compiler about something defined elsewhere (or later).
- In fact, you can use the keyword extern with a prototype



- But, it doesn't change anything, so most people don't.

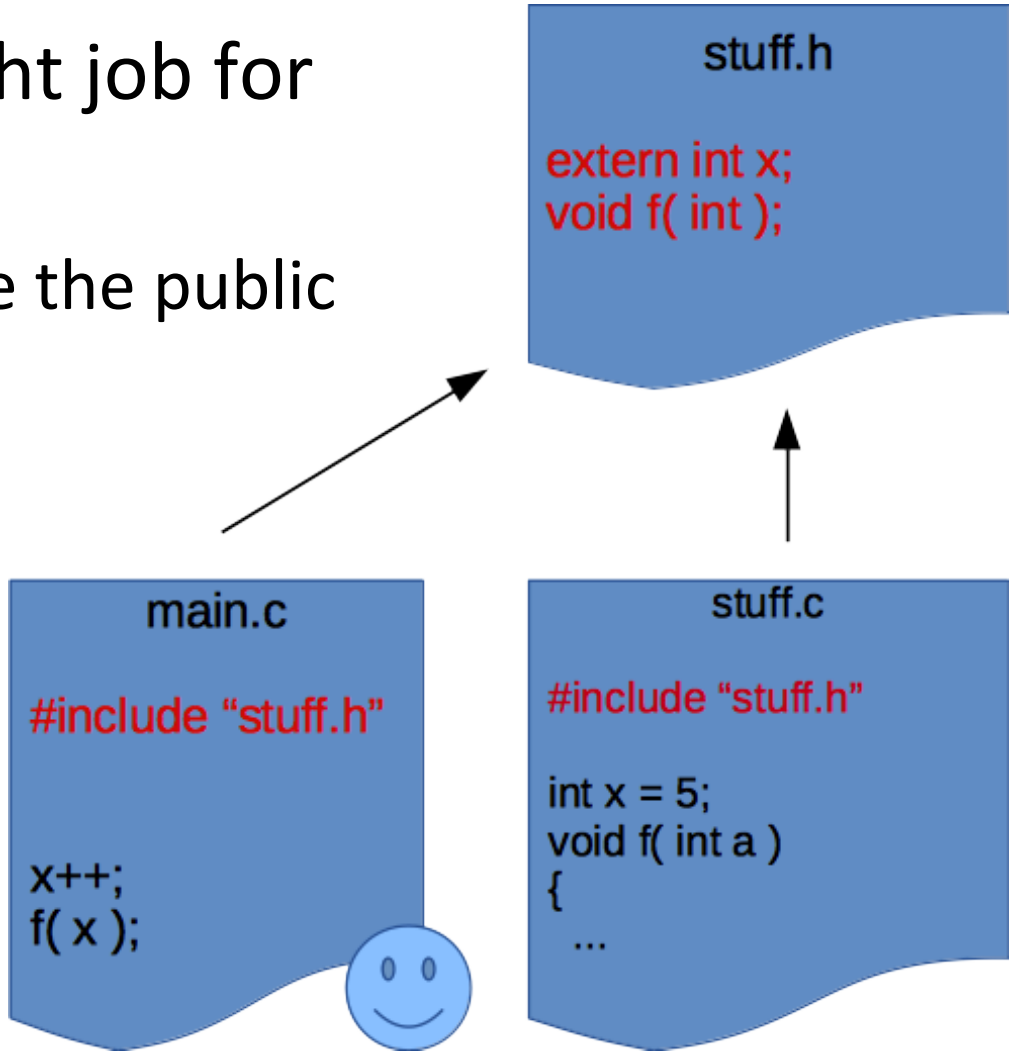
Using Extern Declarations

- We can use an extern variable declarations and prototypes to access symbols (variables and functions) defined elsewhere.
  - This makes the compiler happy when it sees us use them.
  - Then, the linker connects each use with its memory location.

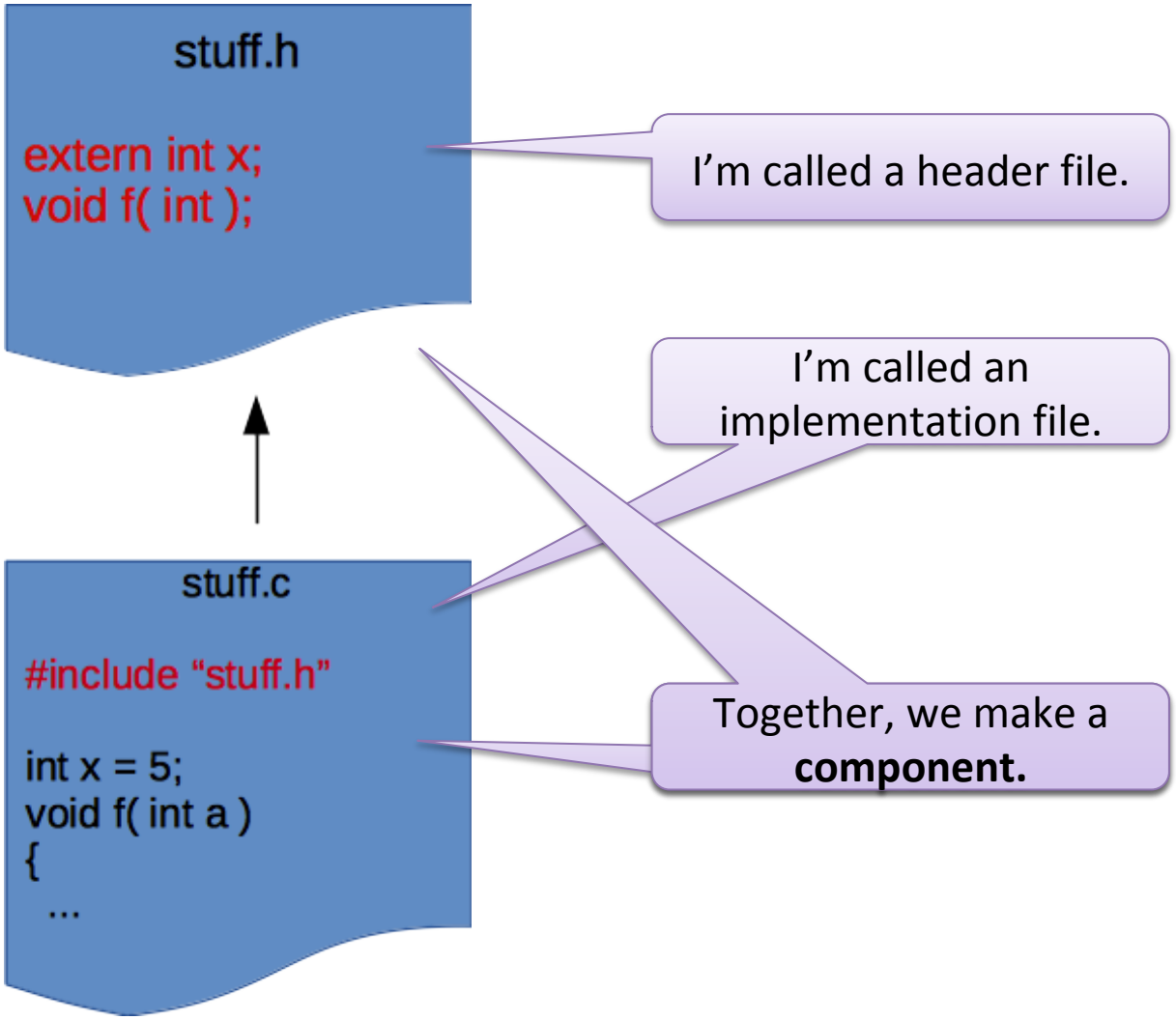


Using Headers

- Really, this the right job for a **header file**.
  - It should advertise the public interface for an **implementation file**.

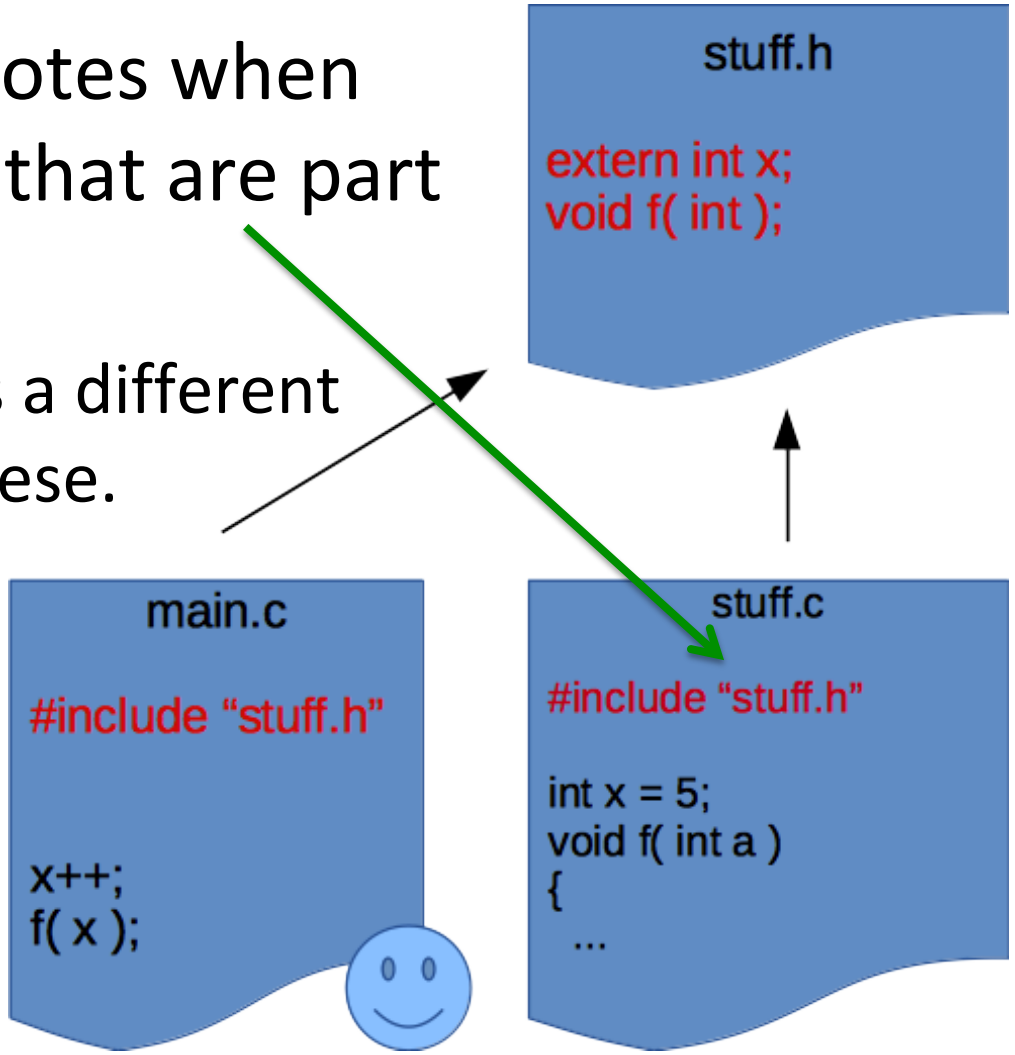


Components


stuff.h
extern int x;
void f(int);
I'm called a header file.
I'm called an implementation file.
stuff.c
#include "stuff.h"
int x = 5;
void f(int a)
{
...
Together, we make a **component.**

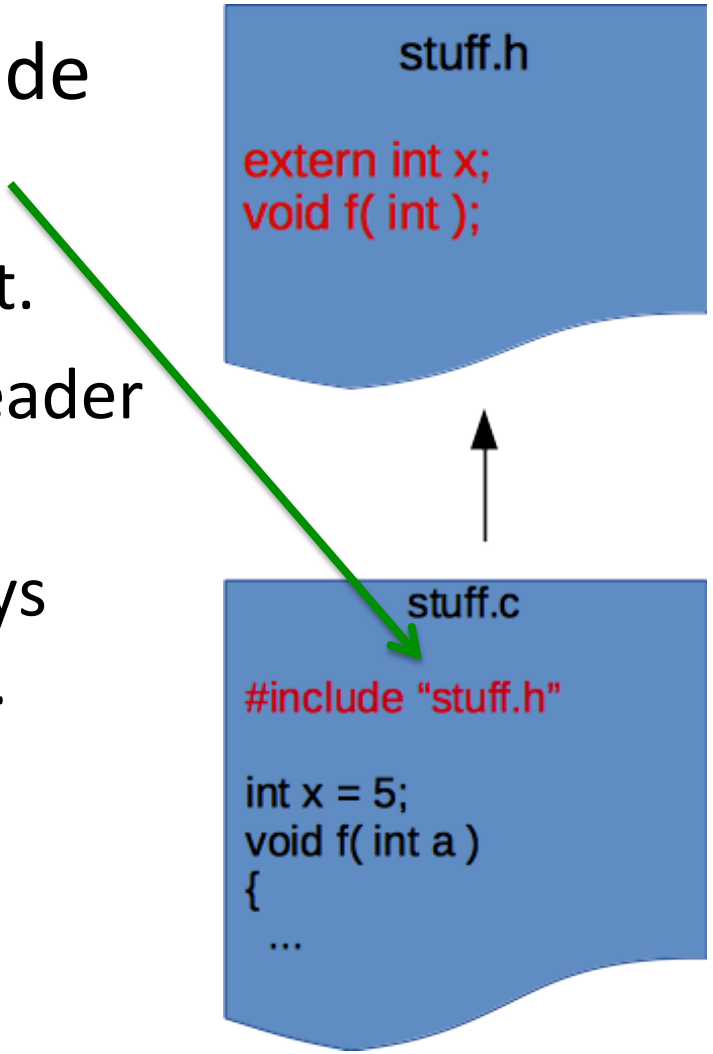
Things to Notice

- We use double quotes when including headers that are part of our project.
  - The compiler uses a different search path for these.



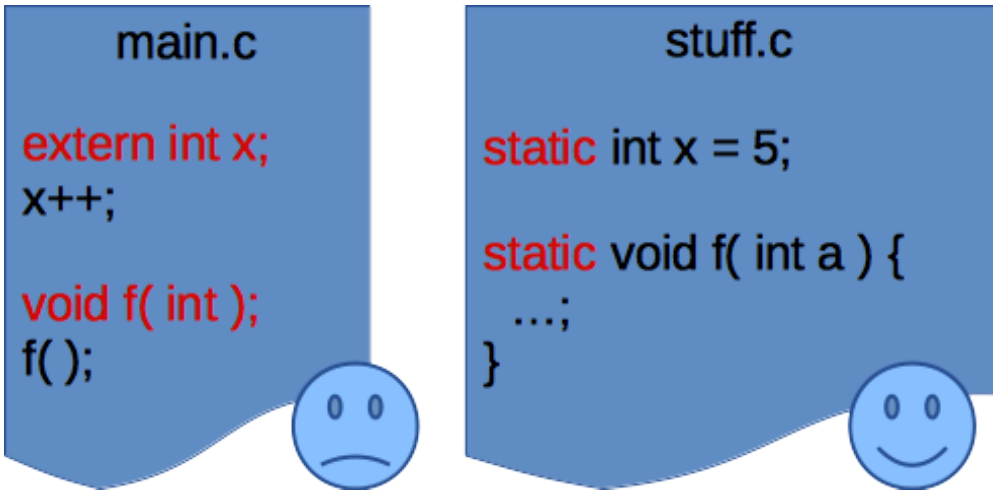
Things to Notice

- It's a good practice to include your own header file first.
  - You may not actually need it.
  - … but it makes sure your header compiles cleanly.
  - … and its declarations always match what you're defining.



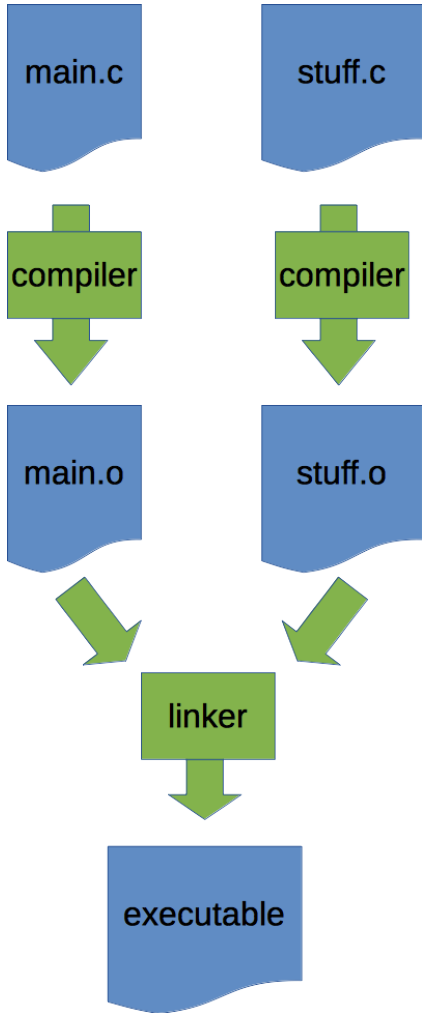
Keeping Things Private

- There's another use of the `static` keyword.
  - For global variables, it marks them as invisible to the linker.
  - This works for functions also.
- We say it gives these symbols *internal linkage*
  - Good for symbols private to one component.
  - Like a private static field in Java.



Building a Program

- After compilation, the *linker* merges all our components into one program.
  - It unifies their address spaces into one coherent space
  - It resolves external references among components
- Even if we only have one component, we still need to link with any libraries we use.



Using the Linker

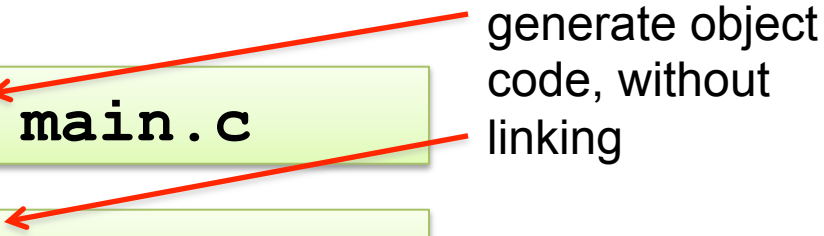
- We can compile and link all at once.

```
> gcc -Wall -std=c99 main.c stuff.c
```

- Or, we can generate intermediate object files.

```
> gcc -Wall -std=c99 -c main.c
```

```
> gcc -Wall -std=c99 -c stuff.c
```



- Then link them together.

```
> gcc main.o stuff.o -o main
```

Compile-Time Errors

- The compiler will complain if it sees you using a symbol it's never heard of.

```
...
f( x );
...
```

```
eos$ gcc -Wall -std=c99 program.c
program.c: … warning: implicit declaration of function ‘f’ …
```

This is a complaint from the compiler.

Link-Time Errors

- With a prototype, the compiler won't complain.
- But, the linker will if it can't find a definition for the function.

```
...
f( x );
...
```

```
eos$ gcc -Wall -std=c99 program.c
test.c:…: undefined reference to `f'
```

This is a complaint from the linker.

Meet nm

- With the nm command, we can look inside an object or library.

```
eos$ nm main.o
                 U f
0000000000000000 T main
                 U x
```

I'm defined in main.o

But not us.

```
$ nm stuff.o
0000000000000000 T f
                 U printf
0000000000000000 D x
```

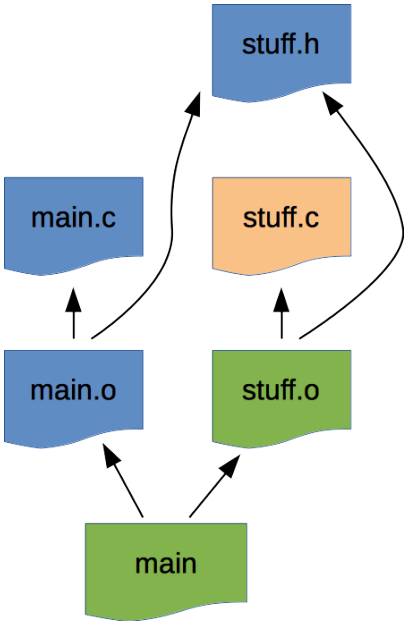
OK, the linker will hook the uses of these symbols up with their definitions.

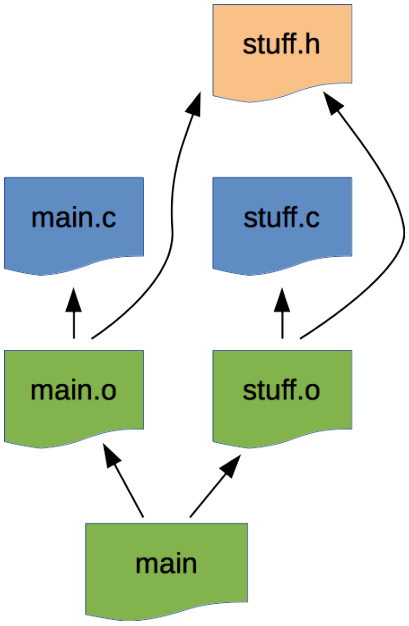
Meet Make

- Consider, our project consists of multiple files, some are source files and others are created during a build
- To build a project, we need to know:
  - the various parts of the project
  - what *dependencies* exist between files we build and the files they depend on
  - *rules* to rebuild files when a dependency is changed
- Make gives us a way to store and use this information:
  - Makefile : A text file format dependency and build instructions
  - make : A tool for following these instructions

Thinking about Dependencies

- A project's dependencies tell us what needs to be re-built if a source file changes.





Makefile Syntax

- Makefiles consist mostly of *rules*.
  - They give a *target*, something that can be built.
  - A list of *prerequisites*, indicating what the target needs and when it must be rebuilt.
  - A list of shell commands, instructions for rebuilding the target from its dependencies.

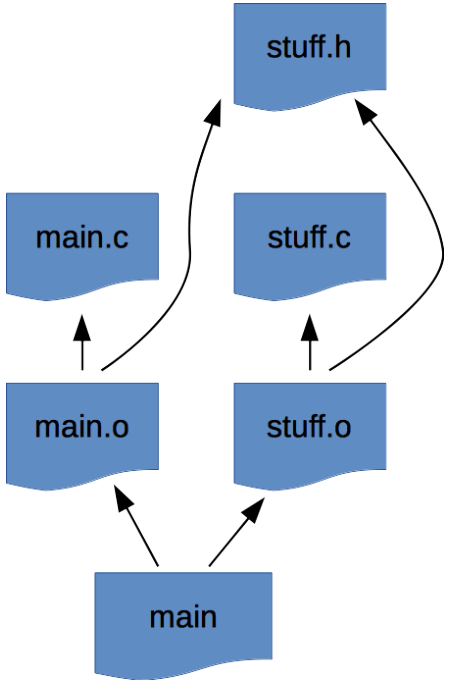
```
target: prereq1 prereq2 ...
        shell-cmd1
        shell-cmd2
        ...
```

Command lines all start with a hard tab.

A Simple Makefile

- A rule for each target
  - Each with prerequisites
  - And a shell command to rebuild the target

```
main: main.o stuff.o
    gcc main.o stuff.o -o main

main.o: main.c stuff.h
    gcc -Wall -std=c99 -c main.c

stuff.o: stuff.c stuff.h
    gcc -Wall -std=c99 -c stuff.c
```



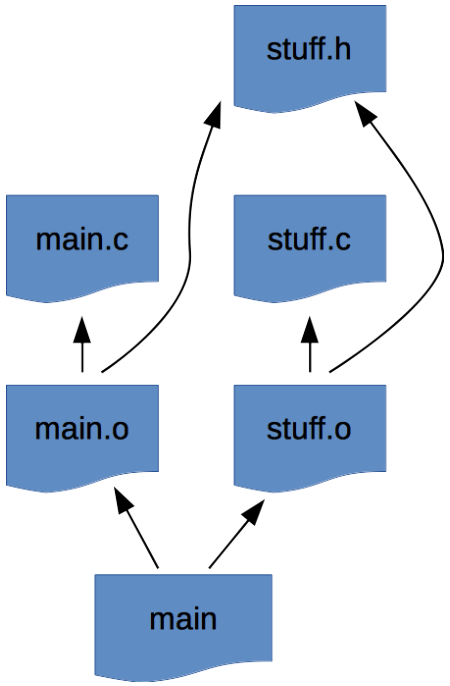
Making Sense of a Makefile

To build this

… first you must build these

… and here's how.

```
main: main.o stuff.o
    gcc main.o stuff.o -o main

main.o: main.c stuff.h
    gcc -Wall -std=c99 -c main.c

stuff.o: stuff.c stuff.h
    gcc -Wall -std=c99 -c stuff.c
```

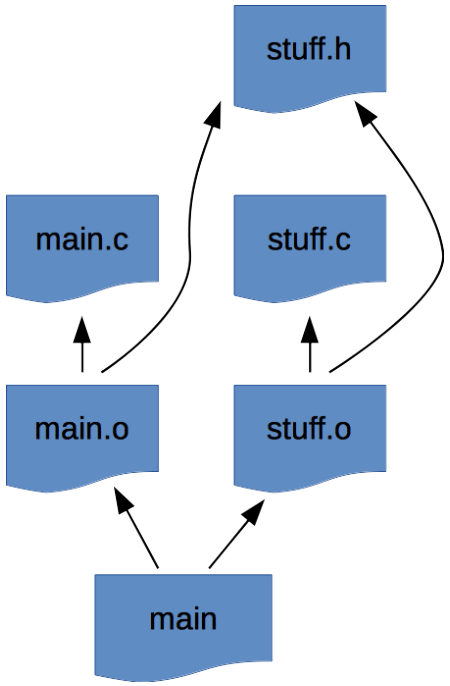


Making Sense of a Makefile

First target is the default.

You can have blank lines, and comments.

```
# This is the default target
main: main.o stuff.o
    gcc main.o stuff.o -o main

# Building each object file
main.o: main.c stuff.h
    gcc -Wall -std=c99 -c main.c

stuff.o: stuff.c stuff.h
    gcc -Wall -std=c99 -c stuff.c
```



Using Makefiles

- You can specify the target to make.

```
$ make
gcc -Wall -std=c99 -c main.c
gcc -Wall -std=c99 -c stuff.c
gcc main.o stuff.o -o main
```

- make is lazy. It only rebuilds what's needed.

```
$ make
make: `main' is up to date.
```

- We can force it to rebuild.

```
$ touch main.c
$ make main.o
gcc -Wall -std=c99 -c main.c
```

This updates the modification time on a file.

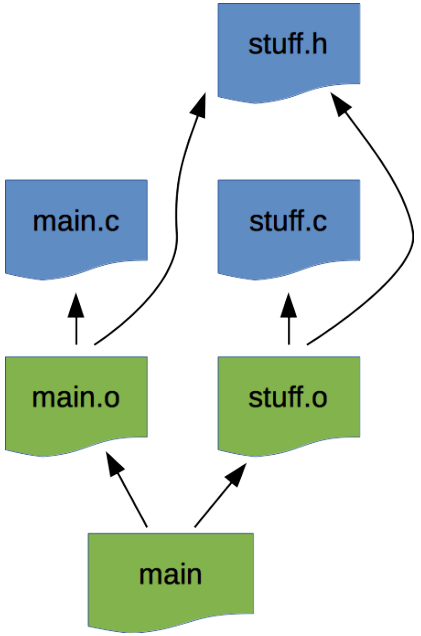
You can choose your own target.

A “Simplified” Makefile

- We can take some shortcuts, but make won’t be able to do as good a job.

```
main: main.c stuff.c stuff.h
    gcc -Wall -std=c99 main.c stuff.c -o main
```

- This makefile will work, but it won’t work as well.



Building A Better Makefile

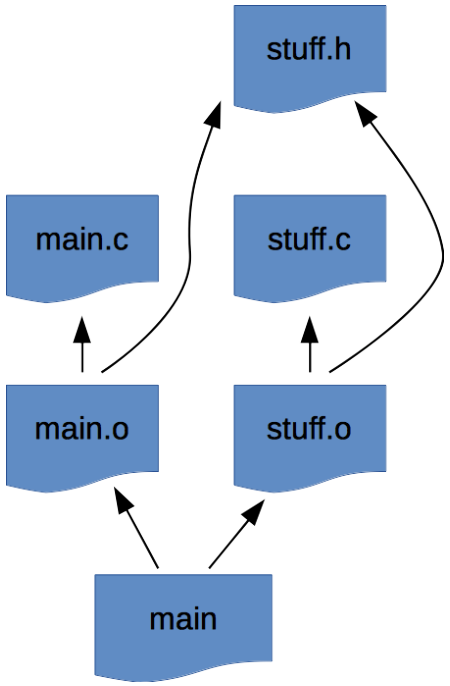
- Make has default rules for building common targets from sources
  - Target **`abc.o`** usually depends on **`abc.c`** with **`gcc -c`** used to build it
  - Objects can be linked to build executable targets
- Make is happy to tell you its default rules.

```
$ make -p -f/dev/null
```

- Using these, we can make a new makefile almost as good as the first.

```
main: stuff.o
main.o: stuff.h
stuff.o: stuff.h
```

Default rules for **everything**.



make Variables

- Make supports variables
  - To configure the default rules
  - And, whatever else you want

```
SRC_FILES = one.c two.c three.c
ALL_FILES = $(SRC_FILES) $(OBJ_FILES) $(EXECUTABLES)
```

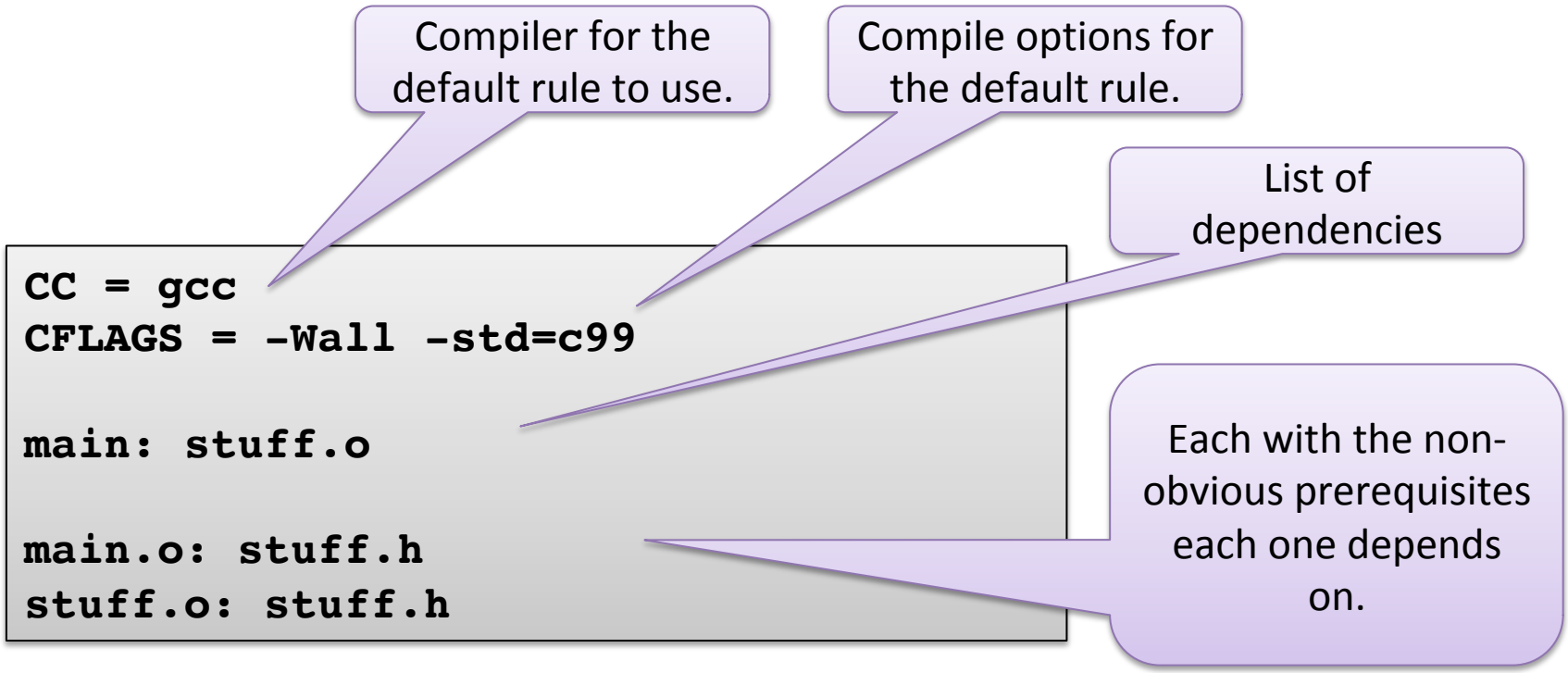
Set a variable.

Get a variable's value.

- The default rules use variables like:
  - CC : the default compiler to use
  - CFLAGS : the default flags to compile with
  - LDLIBS : options for the linker (libraries, e.g., -lm)

Makefiles with Variables

- Now, we can write something that looks like a decent makefile:



More About Make

- Make supports syntax to let you:
  - Define targets that don't build anything
    For example: make clean
  - Choose a particular makefile to use
    For example: make -f Makefile2
  - Define your own reusable rules
  - Decide what to do if a build command fails
- Oh, and I once had a job interview with Stu Feldman, the guy who created make.

On Beyond Make

- Make captures some important ideas, but it has some disadvantages
  - Build instructions in shell hurt portability
- Other tools perform the same kind of task
  - CMake : platform independent, can generate makefiles for various platforms
  - Ant : built in Java, configuration in XML
  - Maven : configuration in XML, dependency management
  - Lots more