Version 2.19

# UCSD CSE 30 Section B

## Computer Organization and Systems Programming

## Lecture 3

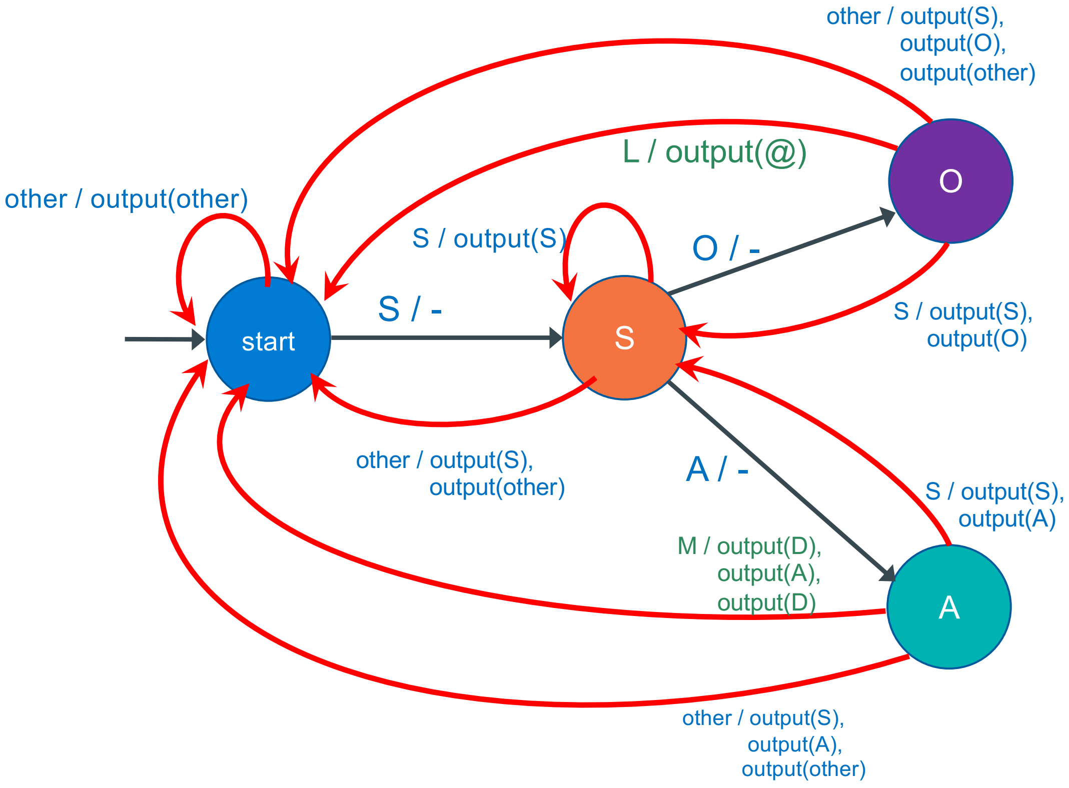
DEC PDP 11/45 - 1973

Keith Muller

# Attendance code

# Merging DFA's – 3 (Finished)

This DFA replaces SOL with a @
and This DFA replaces SAM with DAD



x

# Quick Look: Character and String Literals (more later)

- Usually used to store characters – thus things like file names
- char literals: a single (1) character **inside** a set of **single quotes** 'a'
- string literals: 0 or more characters inside a set of **double quotes** "string"

```
char x = 'a';           // 'a' is a character literal

printf("Hello World!"); // "Hello World!" is a string literal

char a1[] = "xyz";     // char array initialized with contents of a string literal

char b[] = "";         // empty string
```

- Problem: How do you place a **non-printable character** like a **newline** in a literal?
  - The **following are not legal** in C as a **newline** in a source file represents a statement delimiter (white space) in C

```
char x = 'a
';
```

```
printf("Hello World!
");
```

  - Solution: C has a special **line continuation character **

# There are three different uses for \ in C

1. Line continuation sequence a \ followed by zero or more whitespace ending in a newline at the end of a **source line**

Use only when no other choice

```
char a[] = "string: Hello \
World";
```

Poor style use a block comment

```
// line comment \
 rest of line comment
```

**Not needed** do not do this

```
x = x +\
5;
```

2. How do you put a single ' in a character literal or a single " inside a string literal?

- You use an **escape character ** which escapes the special meaning (if any) of the next character inside a character or a string literal

```
char a = '\''; // char: '
```

```
char b = '\\'; // char: \
```

```
char c = '"'; // char: "
```

```
char d[] = "ab\""; // string: ab"
```

```
char e[] = "ab\\"; // string: ab\
```

```
char f[] = "ab'"; // string: ab'
```

```
char a[] = "a "string"";  // syntax error ; expected
char a[] = "a \"string\""; // ok
```

| char sequence | Description |
|---|---|
| `'\\' or "\\"` | `\ char` |
| `'\'' or "\'"` | `single quote` |
| `'\"' or "\""` | `double quote` |

x

# There are three different uses for \ in C - continued

3. You can embed characters with a special meaning inside a (char or string) **literal** using a two-character sequence starting with a \ followed by a single character

- This is typically used for characters that are "non-printable"
- Here are some examples:

| char sequence | Description |
|---|---|
| `'\n'` or `"\n"` | newline char |
| `'\r'` or `"\r"` | carriage return |
| `'\t'` or `"\t"` | tab char |
| `'\b'` or `"\b"` | backspace |
| `'\0'` or `"\0"` | null char |

```
printf("\n\nHello World!\n\n");
```

```
printf("\n\nHello\tWorld!\n\n");
```

# Characters In C

\0 in c encodes a null

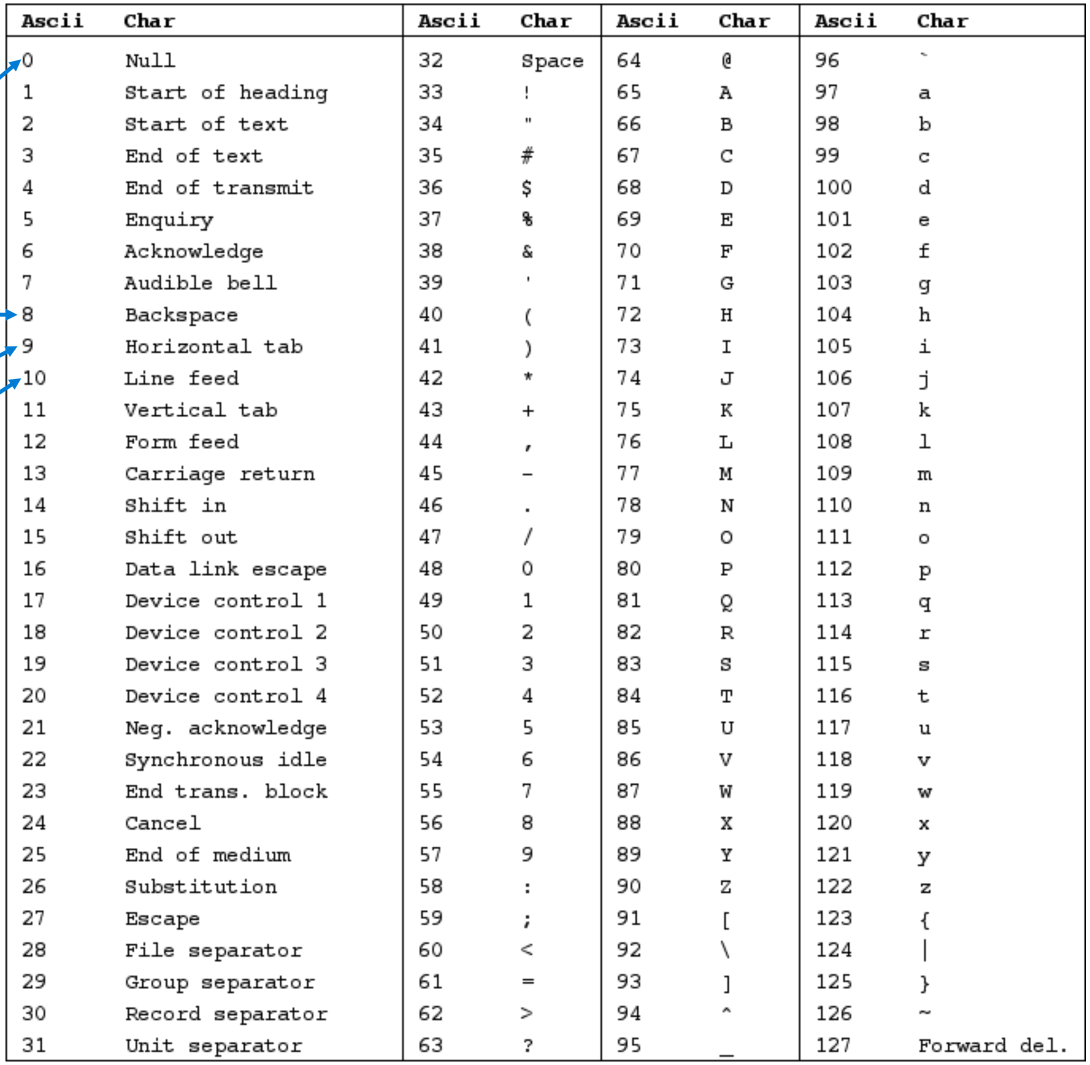
\b in c encodes a backspace

\t in c encodes a horizontal tab

\n in c encodes a linefeed

**Ascii column: decimal integers**

**ASCII Chars are 0-127**
**(stored in 8 bits)**
**Many of the values**
**are not "printable"**



| Ascii | Char | Ascii | Char | Ascii | Char | Ascii | Char |
|---|---|---|---|---|---|---|---|
| 0 | Null | 32 | Space | 64 | @ | 96 | ` |
| 1 | Start of heading | 33 | ! | 65 | A | 97 | a |
| 2 | Start of text | 34 | " | 66 | B | 98 | b |
| 3 | End of text | 35 | # | 67 | C | 99 | c |
| 4 | End of transmit | 36 | $ | 68 | D | 100 | d |
| 5 | Enquiry | 37 | % | 69 | E | 101 | e |
| 6 | Acknowledge | 38 | & | 70 | F | 102 | f |
| 7 | Audible bell | 39 | ' | 71 | G | 103 | g |
| 8 | Backspace | 40 | ( | 72 | H | 104 | h |
| 9 | Horizontal tab | 41 | ) | 73 | I | 105 | i |
| 10 | Line feed | 42 | * | 74 | J | 106 | j |
| 11 | Vertical tab | 43 | + | 75 | K | 107 | k |
| 12 | Form feed | 44 | , | 76 | L | 108 | l |
| 13 | Carriage return | 45 | - | 77 | M | 109 | m |
| 14 | Shift in | 46 | . | 78 | N | 110 | n |
| 15 | Shift out | 47 | / | 79 | O | 111 | o |
| 16 | Data link escape | 48 | 0 | 80 | P | 112 | p |
| 17 | Device control 1 | 49 | 1 | 81 | Q | 113 | q |
| 18 | Device control 2 | 50 | 2 | 82 | R | 114 | r |
| 19 | Device control 3 | 51 | 3 | 83 | S | 115 | s |
| 20 | Device control 4 | 52 | 4 | 84 | T | 116 | t |
| 21 | Neg. acknowledge | 53 | 5 | 85 | U | 117 | u |
| 22 | Synchronous idle | 54 | 6 | 86 | V | 118 | v |
| 23 | End trans. block | 55 | 7 | 87 | W | 119 | w |
| 24 | Cancel | 56 | 8 | 88 | X | 120 | x |
| 25 | End of medium | 57 | 9 | 89 | Y | 121 | y |
| 26 | Substitution | 58 | : | 90 | Z | 122 | z |
| 27 | Escape | 59 | ; | 91 | [ | 123 | { |
| 28 | File separator | 60 | < | 92 | \ | 124 | \| |
| 29 | Group separator | 61 | = | 93 | ] | 125 | } |
| 30 | Record separator | 62 | > | 94 | ^ | 126 | ~ |
| 31 | Unit separator | 63 | ? | 95 | _ | 127 | Forward del. |

x

# Understanding Comments in C (Prep for PA2 and PA3)

- In PA2 (design) and PA3 (program in C), you are going to **write equivalent preprocessor code** to **replace each comment in an input file** with a **single space character (a blank space)** while writing the rest of the input to output unaltered (preserving all newlines)
- **IMPORTANT**: the preprocessor **does NOT perform** any **syntax checking**

```
/* this is /* one block comment */ text outside comment
```

```
// this is // one line comment
text outside comment
```

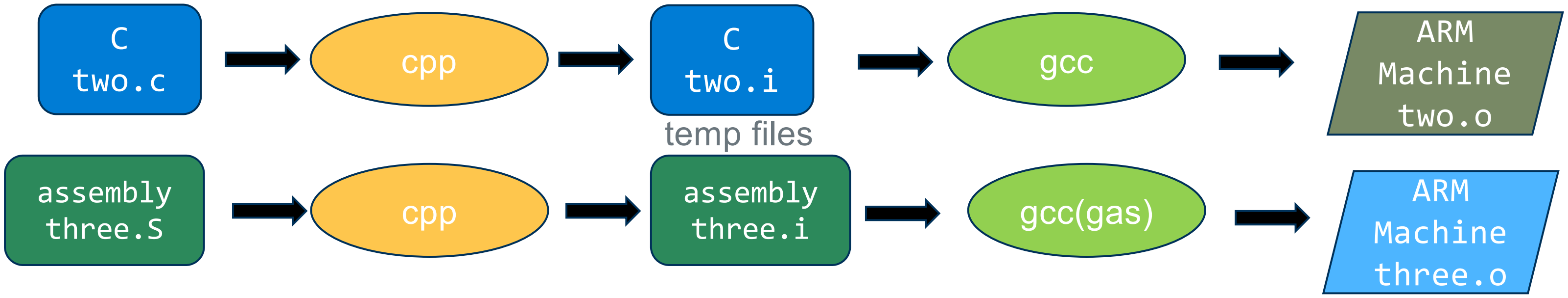
```
/* block comment
// part of block comment not a line comment
yet more block comment
*/ text outside comment
```

```
// line comment /* part of line comment not a block comment */
```

```
//  line comment /* part of line comment not the start of a block comment
oops! text outside of comment, this is not a comment anymore */
```

x

# What is the preprocessor (cpp)?



- **Preprocessing is the first phase** in the compilation (.c files) or assembly (.S files only) process
- The **preprocessor** (**`cpp`**) *transforms* your source code, then **passes it to the compiler** (on .c files) **or the assembler** (on .S files only, not .s files)
  - **cpp is automatically invoked by `gcc`**
- Usually, the input to **cpp** is a C source file (.c) or an assembly source file (.S only) and output from **cpp** is still a C file or assembly file
  - output from cpp is in a temporary .i file (deleted after use)
  - cpp **does not** modify the input source file
- **Common use**: When a **program is divided across multiple source files** (including library files), cpp helps you keep consistency among the files (**one version of the truth**)
  - Examples: Consistent values for a constants, correct function definitions, etc.

x

# Common Preprocessor (cpp) Operations

- **Comments** are ***replaced with a single space*** `/*   */ , // and all newlines are preserved`
  - You will do a design for this in PA2 and program it in PA3
- **Continued lines:** where the **last character in a line is a \** causes the line to be **joined with the next line**
- A ***preprocessor directive:*** commands to cpp to perform an operation (these start with a **#)**
  - `#include <stdio.h>` contents of the file stdio.h is to be *inserted* at that spot in the source file
  - `#define MAX 8`
    - **Does two things: Defines MAX** to be a ***macro** name* **and** assigns it the value 8
      - `#define MINE` just defines MINE to be a macro name with no value (for conditional tests – later)
    - **Convention**: **MACRO** names are in **CAPITAL** letters
  - Macros with values `– cpp replaces MAX with 8` everywhere in the source file

```
#define MAX 8
int main(void)
{
    int x[MAX]; // histogram array

    for (int i = 0; i < MAX; i++) {
       …
    }
     …
}
```

cpp input

```
int main(void)
{
    int x[8];

    for (int i = 0; i < 8; i++) {
       …
    }
     …
}
```

cpp out (only showing macro substitution)

file ex.i

x

# Complexity for programming a preprocessor: Literals may contain what appears to be comments, but are not

```
char x = 'a';                 // 'a' is a character literal

printf("Hello World!");       // "Hello World!" is a string literal
```

```
"/* text */" not a comment but a string literal whose contents looks like a block comment
```

```
"// text" not a comment but a string literal whose contents looks like a line comment
```

```
'/* text */' not a comment but a character literal (not legal, but that is the compilers
             job) whose contents looks like a block comment
```

```
'// text' not a comment but a character literal (not legal, but that is the compilers
          job) whose contents looks like a line comment
```

x

# cpp conditional (and macro) only operations

- You can use **conditional preprocessor tests** (like if-else statements) around blocks of code

  `#ifdef MACRO, #ifndef MACRO, #else, #endif`

- In this use, MACRO is called the guard MACRO ("guards" entry to the following block)

`#ifdef MACRO` if MACRO is defined, then the block is included, otherwise the `#else` block (if any) is included

`#ifndef MACRO` if MACRO is NOT defined, then the block is included, otherwise the `#else` block (if any) is included

`#endif` is the end of a block

```
#define MACRO           // defines MACRO  -- #define MACRO 8 defines macro and assigns a value of 8

#undef MACRO            // undefines MACRO
```

```
#define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    int x[8];
    ….
    return;
}
```

```
// #define VERS1
#define MAX 8
// file ex.c
void func(void)
{
#ifdef VERS1
    int x[MAX];
#else
    short x[MAX];
#endif
    ….
    return;
}
```

after the preprocessor runs

```
void func(void)
{
    short x[8];
    ….
    return;
}
```

X

# First Look at Header Files (also called .h  or "include" files)

- Header file:  a file whose only purpose is to be **#include'd** by the **preprocessor**
  - Contains: **Exported (public) Interface declarations**
    - Examples: function prototypes, user defined types, global variable, macros, etc.
  - Used to import the public interface of another C source file
    - #include its header (interface) file
- **NEVER EVER use cpp to** #include a .c file, a .S or a .s file
- **Convention (strongly enforced):** header files use a .h  filename extension (example:  filename**.h)**
  - **Example**: Source file src.**c** exported (public) interface is in the header file src.**h**
- How to specify the file to be **#include'd**
  - <system-defined>  are system header files (typically located under /usr/include/…)

    ```
    #include <stdio.h>   // located in /usr/include/stdio.h
    ```
  - "programmer-defined" header files usually in a relative Linux path (see –I  flag to gcc)

    ```
    #include "else.h"    // looks in the current directory first
    ```
- **Convention:**  #include  directives are usually placed near the top of a source file above any code

# Compilation Process Operations

```
#include <stdlib.h>
#include <stdio.h>

// A simple C Program
int
main(void)
{
    printf("Hello World!\n");
    return EXIT_SUCCESS;
}
```

preprocessor: inserts and processes the contents of files here.
Inserts: Function protype for printf (later in course)
macro value for EXIT_SUCCESS
File locations: /usr/include/stdio.h & /usr/include/stdlib.h

preprocessor: replaces the line Comment with one blank

compiler generates assembly code to call the library function printf() and pass the string "Hello World!"

cpp: replaces EXIT_SUCCESS with 0 on Linux

compile: **gcc –Wall –Wextra prog.c -o prog**

1. cpp first processes the file (cpp is called by gcc)
2. Compiler (gcc) compiles main to assembly
3. Assembler (gas – called by gcc) translates the assembly to machine code
4. Linker (ld) merges the machine code for printf() (from a library) with your programs machine code to create the executable file **prog** (machine code)
    - -o specifies the name of the executable (default: **a.out**)

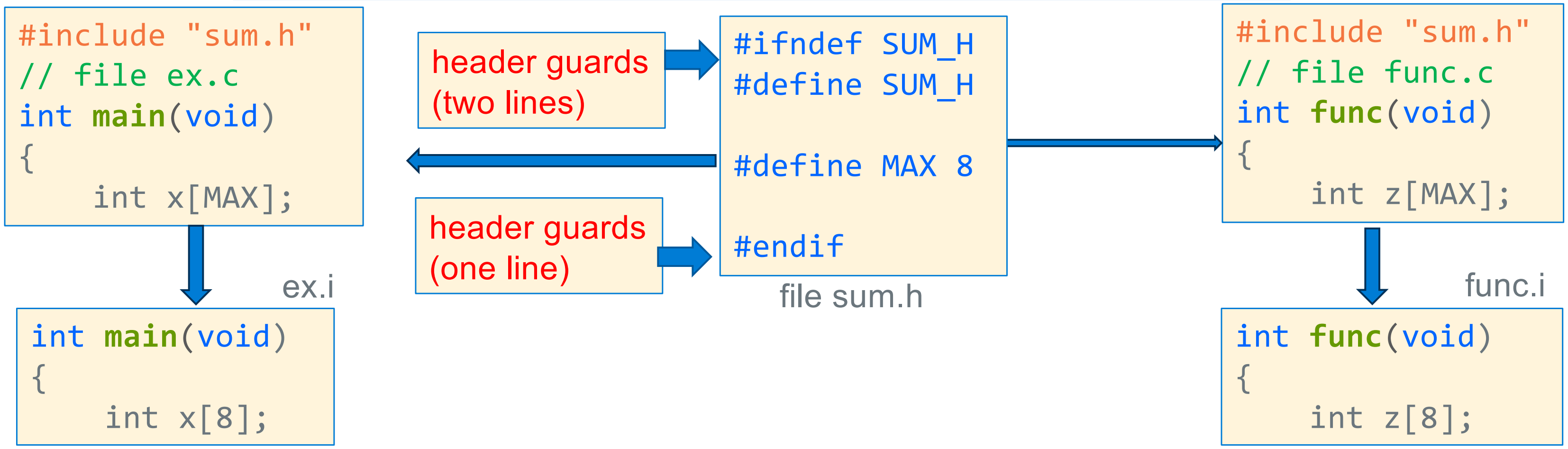
x

# cpp conditional tests: header guards

- Header guards ensure that only **one copy of a .h file** is **included in a source file**
- **A Convention**: header guard (macro) NAME (all capital letters) is created as follows:
  - use the **filename of header file but** in all caps
  - **replace** the **period in** header file **name** with an _
  - Example: file sum.h header guard macro name is SUM_H

- How do you use "header guards" in your code?

```
#ifndef NAME_H            // first line in the file
#define NAME_H
          . . .
#endif                    // last line in the file
```



```
#include "sum.h"
// file ex.c
int main(void)
{
    int x[MAX];
```

header guards (two lines)

header guards (one line)

```
#ifndef SUM_H
#define SUM_H

#define MAX 8

#endif
```

file sum.h

```
#include "sum.h"
// file func.c
int func(void)
{
    int z[MAX];
```

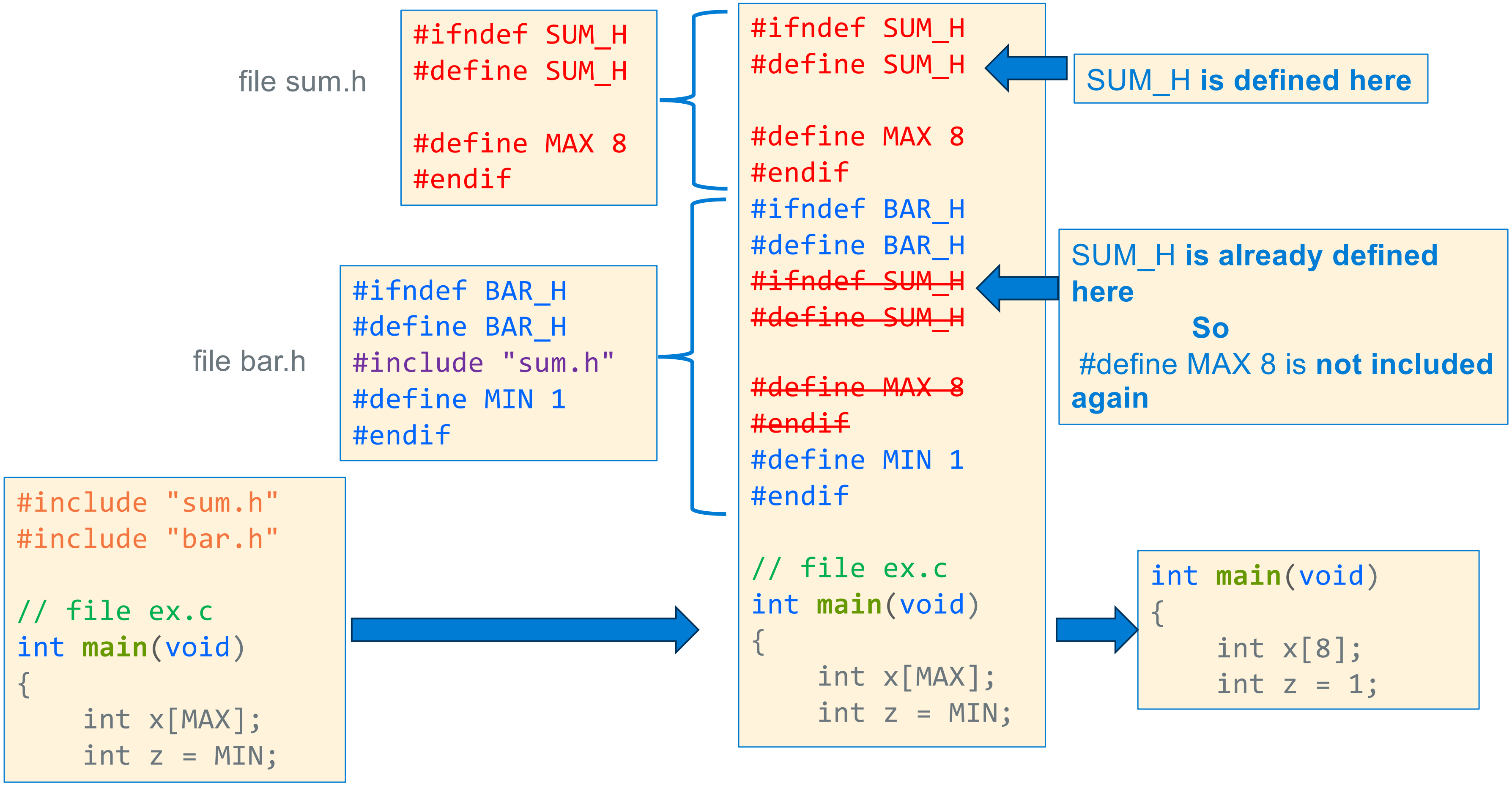
ex.i

```
int main(void)
{
    int x[8];
```

func.i

```
int func(void)
{
    int z[8];
```

# Why header guards are needed



file sum.h

```
#ifndef SUM_H
#define SUM_H

#define MAX 8
#endif
```

file bar.h

```
#ifndef BAR_H
#define BAR_H
#include "sum.h"
#define MIN 1
#endif
```

```
#include "sum.h"
#include "bar.h"

// file ex.c
int main(void)
{
    int x[MAX];
    int z = MIN;
```

```
#ifndef SUM_H
#define SUM_H

#define MAX 8
#endif
#ifndef BAR_H
#define BAR_H
~~#ifndef SUM_H~~
~~#define SUM_H~~

~~#define MAX 8~~
~~#endif~~
#define MIN 1
#endif

// file ex.c
int main(void)
{
    int x[MAX];
    int z = MIN;
```

SUM_H **is defined here**

SUM_H **is already defined here**

**So**

#define MAX 8 is **not included again**

```
int main(void)
{
    int x[8];
    int z = 1;
```

x