

Version 2.21

UCSD CSE 30 Section B

Computer Organization and Systems Programming

C Part 1

Keith Muller

DEC PDP 11/45 - 1973





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;
```

output of cpp

x

Background: What is a Definition?

- **Definition:** creates an instance of a *thing*
- There **must be exactly one** definition of each *function or variable* (no duplicates)
- **Function definition (compiler actions)**
 1. **creates code** you wrote in the functions body
 2. **allocates** memory to store the code
 3. **binds** the function name to the allocated memory
- **Variable definitions (compiler actions)**
 1. **allocates memory:** **generate code** to **allocate space** for local variables
 2. **initialize memory:** **generate code** to **initialize the memory** for local variables
 3. **binds (or associates)** the variable name to the allocated memory

C Function Definitions - 1

- **C Functions are not methods**
 - no classes, no objects
- **C function definition**
 - returns a value of returnType
 - **zero** or more **typed** parameters
- Every program must have initial (start) function: `int main(int argc, char **argv)`
- `main()` is the **first function in your code** to run/execute
 - `main()` is **not the first function** to run in a Linux process, it is the **C runtime startup code**
 - later in course
 - You should never **make a call to `main()`** from your code

```
returnType fname(type param1, ..., type paramN)
{
    // statements
    return value;
}
```

function definition

```
// returns: sum of integers from 1 to max
int
sum(int max) // function definition
{
    int i, sum = 0; // variable definition

    for (i = 1; i <= max; i++) {
        sum += i;
    }



    return sum;
}
```

C Function Definitions - 2

remember this is a pre-processor (cpp) macro
it is not a variable, it is a "substitution"

- A function of type `void` does not return a value
- A `void` parameter or an **empty parameter (argument) list** specifies this is a function with **no parameters**
 - A common practice is to use the keyword `void` to specify an empty or an **ignored** parameter list
- At runtime, **function parameters** are **evaluated**, then the resulting **values are COPIED** to a memory allocated for the parameter (like a local variable) – this is very important to know
 - So, functions are **free to change** parameter values in their body without side effect to the calling function
 - **C Parameter passing** is called: **call by value**

```
// prints sum of integers 1 to MAX  
#define MAX 8
```

```
int  sum(void)  // or sum()  
{  
    int i, total = 0;  
  
    for (i = 1; i <= MAX; i++) {  
        total += i;  
    }  
  
    return total;  
}
```

C Function Definitions - 3

- In standard C, functions **cannot be nested (defined)** inside of another function (called *local functions in other languages*)

```
int outer(int i)
{
    int inner(int j) // do not do this, not in standard c
    {
    }
}
```

- Assignment inside conditional test with a function call** (this is very common!)

```
if ((i = SomeFunction()) != 0)
    statement1;
else
    statement2;
```

assignment returns the value that is placed into the variable to the **left of the = sign**, then the test is made

Textbook Over-ride: Linux Return Value Convention

- In your code, `main()` is the first function to start to execute and *usually* the last
- **Linux** uses a **convention** on **signaling errors** at process termination to the command line "shell"
 - Remember checking return values in CSE15L scripts?
 - It is the value often associated with the `return` statement from `main()`
- **In this class**, **always** use the **Linux standard return codes** as defined in `<stdlib.h>` when returning from `main()` or exiting your program

`EXIT_SUCCESS` *// program completed ok; usually 0*

`EXIT_FAILURE` *// program completed with error; non-zero value*

`return EXIT_SUCCESS;`

Setting program termination return (status) values

Indicating your program
operated correctly

```
#include <stdio.h>
#include <stdlib.h>
int
main(void) {
    /* Your code here */
    /* code was successful */
    return EXIT_SUCCESS;
}
```

Indicating your program
operated incorrectly/errors

```
#include <stdio.h>
#include <stdlib.h>
int
main(void) {
    /* Your code here */
    /* a failure occurred */
    return EXIT_FAILURE;
}
```

Background: What is a Declaration?

Declaration: describes a *thing* – specifies types, **does not create** an instance

- **Each declaration** has an associated *identifier* (the name)
 1. **Function prototype:** describes how to **write the code to call a function** defined elsewhere
 - **Identifier** is the **function name**
 1. Describes the **type of the function return value**
 2. Describes the **types of each of the parameters**
 2. **Variable declaration:** describes how to **write the code to use a variable** in a statement
 - **Identifier** is the **variable name**
 - Describes the **type of a variable** that is **defined elsewhere**
 3. **Derived and defined type description**
 - **Identifier** describes the derived/defined type
 - struct, arrays, plus others (covered later)
- An **identifier** may be **declared multiple times**, but **only defined once**
- A **definition** is also a **declaration** in C

Definitions and Declarations Use in C

You must **declare a function or variable before you use it**

- **Warning:** Use before declaration will implicitly cause types to default to be of type **int**

sumit() is BOTH defined and declared here

Independent Translation Unit: the granularity (unit) of source which is compiled or assembled

Default **Definition** and **declaration range of validity:**

1. **Restricted** to the file (translation unit) where they are located **and**
2. **Start at the point** of definition or declaration in the file, stopping at the **end of the source file (translation unit)**

Observation: Requiring function order in a file is a pain....

- (1) sum() must be defined in the same source files
- (2) sum() appear before it is used by main()

Question: How do we remove this limitation?

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 8
int sumit(int max)
{
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
// observe sumit() is declared above main()
int main(void)
{
    printf("sum: %d\n", sumit(MAX));
    return EXIT_SUCCESS;
}
```

i, sum, are both defined and declared here

sumit() is used here

Function Prototypes: How to Declare a Function

Function prototype is how we **declare a function** in C

```
returnType fname(type_1, ..., type_n); // function prototype
```

- **Function prototype** is *function definition header* followed by a single semicolon (;) **NO code block**
- **Declares the function valid** from that **point** in the source file, **stopping at the end of THAT FILE!**

- C requires the **function declaration to be seen in the source file before use**

The **function prototype** for sum() enables:

1. body of sum() to be either after main() in the same source file **or**
2. body of sum() to be in a different source file (but this will use of interface files – in a few slides)

Common practice: Function prototypes in a .C file are usually placed at the top the file

code block

```
#include <stdlib.h>
#define NUM 100
int sum(int); // function declaration starts here
int main(void)
{
    sum(NUM); // rest of code not shown
    return EXIT_SUCCESS;
}
int sum(int max) // function definition is here
{
    int i, sum = 0;
    for (i = 1; i <= max; i++) {
        sum += i;
    }
    return sum;
}
```

C and Scope Review

- **Scope:** Range (or the extent) of instructions over which a name/identifier is allowed be referenced by C instructions/statements
 1. **File Scope:** Range is within a single source file (**translation unit**)
 2. **Block Scope:** Range is within an enclosing block (for variables only)

```
int global;                // global variable with file scope

void foo(int parm)         // function foo with file scope
{                           // parameter parm block scope begins
    int i, j = 5;          // function body (block) begins
    for (int k = 0; k < 10; i++) { // variables with block scope
        // some code        // inner block scope
    }
}                           // function body ends
```

Nested Scope

- **Nested Scope:** When two different variables have the same name are in scope at the same time, the declaration (*remember definitions are also declarations*) that appears in the inner scope hides the declaration that appears in an outer scope

```
void funcA(int n)           // scope of the function parameter 'n' begins
{                           // the body of the function begins
    ++n;                   // 'n' is in scope and refers to the function parameter
    // int n = 2;          // error: cannot redeclare identifier in the same scope

    for(int n = 0; n < 10; ++n) { // scope of loop-local 'n' begins
        printf("%d\n", n);        // prints 0 1 2 3 4 5 6 7 8 9
    }                             // scope of the loop-local 'n' ends

    printf("%d\n", n);          // the function parameter 'n' is back in scope
                                // prints the value of the parameter

}                               // scope of function parameter 'n' ends
```

C Variable Storage Lifetime

1. **Static Storage Lifetime:** valid while program is executing
 - Storage allocated **and initialized prior to program start** (implicit default = 0)
2. **Automatic Storage Lifetime:** valid while enclosing block is activated
 - **Storage allocated and is not implicitly initialized (value = unspecified)** by **executing code** when entering scope and **made available for reuse by executing code** when exiting scope
 - It is **not correct to say that automatic storage has been deallocated on exit** (it *might be*) but more often is *still part of your program and may be referenced from the viewpoint of the OS without causing a runtime fault* if you have an address (pointers later in course)
 - **Contents of storage after exiting scope is not changed** (why would C act this way?)
3. **Allocated Storage Lifetime:** valid from point of allocation until freed or program termination
 - Storage allocated by call to an allocator function (malloc() etc.) at runtime and **is not implicitly initialized (value = garbage)** - one allocator does initialize to zero at runtime calloc() – later in course
4. **Thread Storage Lifetime:** valid while thread is executing (not CSE 30)

Variables in C

- **Global variables**
 - **Defined at file scope** (outside of a block)
 - have **static storage duration**
 - global variables **defined without an initial value default to 0** (set prior to program execution start)
 - global variables **defined with an initial value are set at program start**
- **Local (block scope, or automatic) variables** (including function parameter variables)
 - **Defined at block scope** (inside of a block)
 - have **automatic storage duration, with one exception (see below)**
 - block scope variables **defined without an initial value have an unspecified initial value**
 - block scope variables **defined with an initial are set each time by code when the block is entered**
 - All block scope variables **become unspecified at block exit**
- **Variable definitions preceded by the keyword *static*** always have **static storage duration** including variables defined with block scope (when used global variables it restricts scope – later slides)

```
int global;           // global with static storage duration, initial value = 0
int foo(void)
{
    static int s;      // "local" with static storage duration, initial value = 0
    int x;             // "local" with automatic storage duration
}
```


Example:

Block scope (local) static storage duration variables

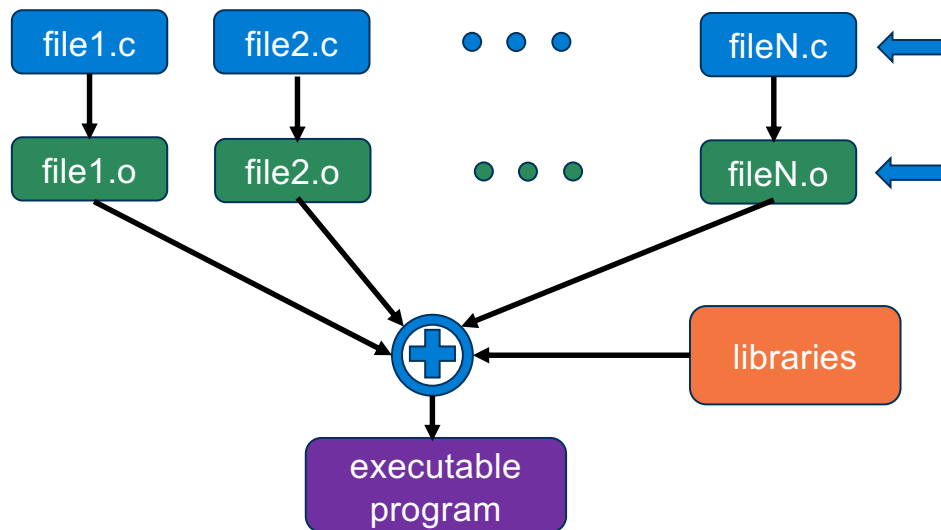
```
#include <stdio.h>
#include <stdlib.h>
#define MAX 5

int foo(void)
{
    static int s; //static storage duration, set to 0 at program start
    return s += 1;
}

int main(void)
{
    for (int i = 0; i < MAX; i++)
        printf("%d ", foo());
    printf("\n");
    return EXIT_SUCCESS;
}
```

```
% ./a.out
1 2 3 4 5
%
```

Real programs are distributed across multiple files



Example: fixing a bug in a existing program

1. You fix bug in just `fileN.c`
2. Only need to recompile `fileN.c` to `FileN.o` (all the other `.o` files are fine)
3. Relink all `.o` files and libraries
4. Test the executable

- **Large programs** in one source file can be very difficult to manage
 - Consider a program with many millions of lines of code
 - And there are 100's developers working on it, changing source parts of the code
 - The program is being rebuilt (compiled/linked) and tested several times a day
- **Approach:** Break a program into individual translation units (source files)
 - **Compile them individually** and then link them together
 - Only need to recompile those source files that have changed

Controlling Linkage Across Files in Multi-File C Programs

- **Linkage** determines whether an object (like a variable or a function) can be referenced **outside the source file it is defined in**
- **External Linkage:** function and variables with external linkage **can be referenced anywhere in the entire program**
 - **Global variables** and **all functions** have external linkage by **default**
 - **Unless explicitly declared, the default type is int for both functions and global variables**
 - **However**, the compiler must know the correct types before the use of a function or a variable, so it is able to generate the correct code
 - **NEVER DEPEND** implicit default typing
 - Use **function prototypes** to **declare functions** before use
 - Use the keyword **extern** to "extend the visibility", **e.g., declare** a global variable before use

```
// example here is at file scope
extern int x;    // declaration
int x = 10;     // definition
```

Controlling Linkage Across Files in Multi-File C Programs

- **Internal Linkage (private):** functions and global with internal linkage can **only be referenced** in the **same source file**
- Global variables and functions can be defined to have **internal linkage** by using the keyword **static** in front of the definition (confusingly another use of the word static)

```
static int global2;  
static int funcB(void) { }
```

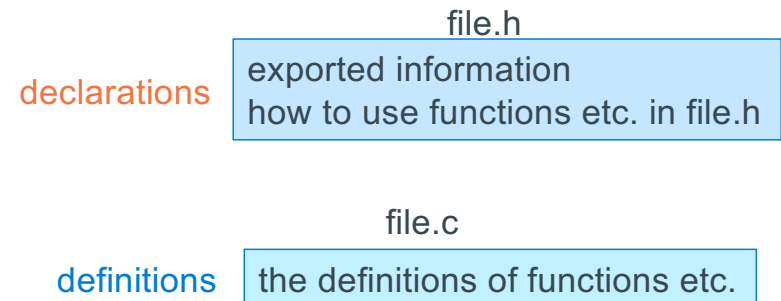
- Use of the keyword **static** in front of a **global variable definition** or **function definition** changes it to **internal linkage** and effectively makes it **private to the file they are defined in** (It cannot be referenced by another file)
- Function definitions in **different files** (translation units) can **re-use the same name** if **at most one has external linkage** (all others must be internal linkage)
- **No Linkage:** function parameters, variables defined inside a block (including a functions body)
 - **Remember:** the keyword **static** in front of a **block scope variable** changes the variable to **static storage duration** (it does not change the linkage)

Linkage Examples

```
int global0 = 1;           // external linkage
static int global2;        // internal linkage to this file, default initial value = 0
int funcA(int x)           // funcA has external linkage; x has no linkage
{
    int y;                 // no linkage, no initial value set (value unspecified)
}
static int funcB(void)     // internal linkage restricted to this file
{ }
```

Creating Public Interface files (header files)

- To enable a **source file** to **use any of the functions**, **global variables**, and **MACROS** defined in another file (separate translation unit)
 - You must create a file that exports all permitted accesses so the compiler can generate the correct code
- **Convention:** For each source file, **file.c**, the **public interface file** is **file.h**
- If a file has no external interfaces, then it does not need a .h file



- **file.h** contains any
 - public preprocessor macros
 - **function prototypes** for the functions defined in the source file, **file.c that you want visible (exported)** for use (called) by functions defined in other source files
 - *global variable declarations (external linkage)*
 - **Do not put any definition statements** in a header file

- **file.c** contains
 - All function and global variable definitions (internal and external linkage)
 - Any private preprocessor macros
 - Any private (internal linkage) function prototypes

Creating Public Interface files (header files)

- Always #include your own declaration files BEFORE any definitions
- compiler will then check that the definition and declarations are consistent



using the public interface

```
// myprog.c
#include <stdlib.h>
#include <stdio.h>
#include "file.h"

// code not shown
int main(void)
{
    // body not shown
}
```

public interface for file.c

```
// file.h
#ifndef FILE_H
#define FILE_H

#define MAX 5

extern int global;

int A(int);
char B(int, int);

#endif
```

```
// file.c
#include <stdlib.h>
#include "file.h"

static int P(char );
    // above: private function prototype

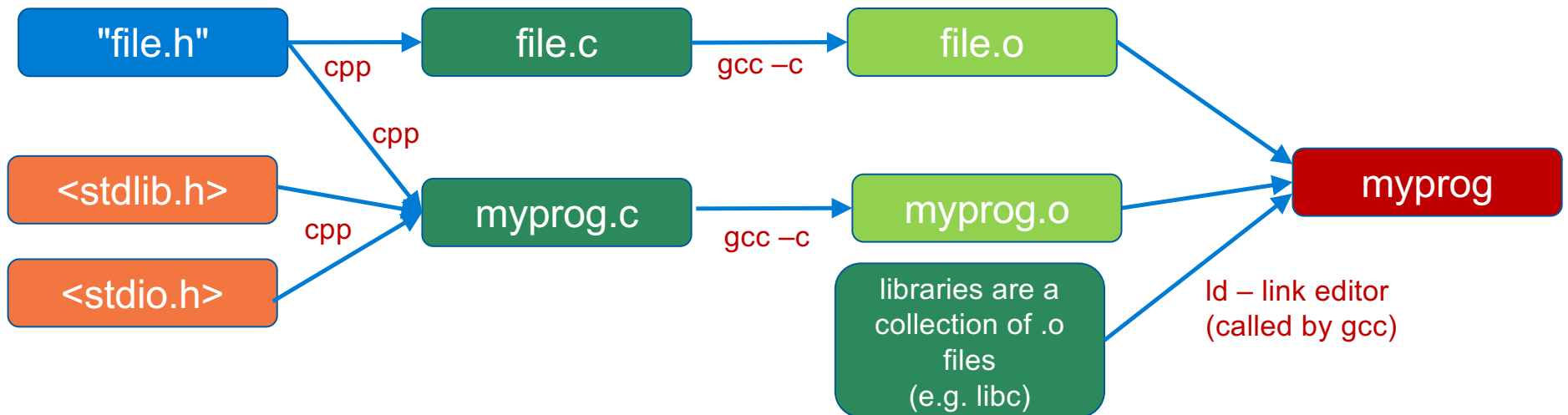
int global;           // initial value is 0
static int private = 1; // private global

int A(int c)
{
    // body not shown
}

char B(int x, int y)
{
    // body not shown
}

static int P(char z)
{
    // body not shown
}
```

Compiling Multi-File Programs (assembly steps not shown)

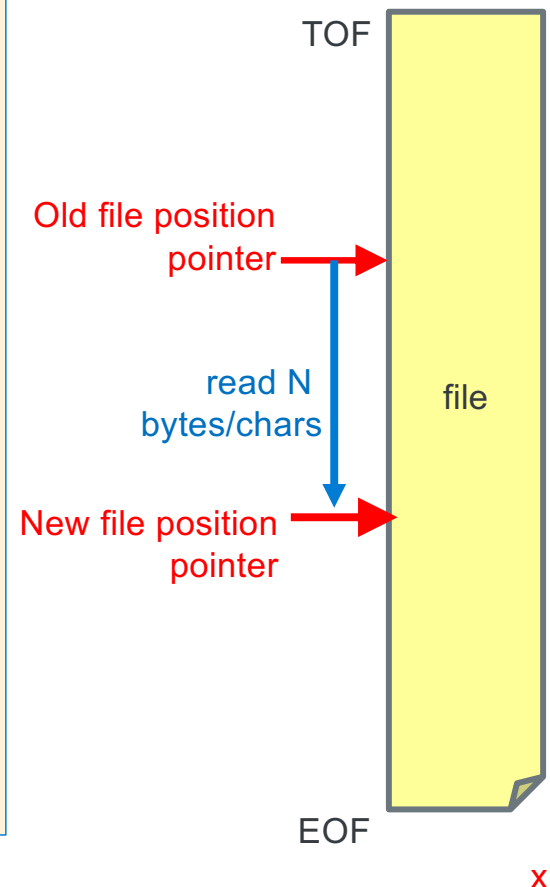


1. compile each .c file independently to a .o object file this requires you use the -c flag to gcc to only compile and assemble and NOT to call the linker yet
`gcc -Wall -Wextra -Werror -c file.c # creates file.o`
`gcc -Wall -Wextra -Werror -c myprog.c # creates myprog.o`
2. link all the .o objects files and libraries (aggregation of multiple .o files) to produce an executable file (gcc calls ld, the linker)
 - The .o's in the libraries are automatically linked in as needed to produce an executable file`gcc -Wall -Wextra myprog.o file.o -o myprog`

C standard I/O Library (stdio) File I/O

File Position Pointer and EOF

- Read/write functions in the standard I/O library *advances* the **file position pointer** from the **top of a file** (before the 1st byte if any) *towards* the **end of the file** after each call to a read/write function
 - **Side effect of call:** file position pointer moves towards the **end of file** by number of bytes read/written
- **standard I/O File position pointer** indicates where in the file (byte distance from the top of the file) the next read/write I/O will occur
- Performing a sequence of read/write operations (without using any other stdio functions to move the file pointer between the read/write calls) performs what is called **Sequential I/O** (sequential read & sequential write)
- EOF condition state may be set after a **read operation**
 - After the last byte is read in a file, additional reads results in a **function return value** of EOF
 - **EOF signals** no more data is available to be read
 - **EOF is NOT a character in the file**, but a condition state on the stream
 - EOF is usually a **#define EOF -1** macro located in the file stdio.h (later in course)



Reference Slides

- Slides in this section are not used in class but contain material that you will find useful

Cast

- Java: demotions are not automatic
C: demotions are automatic
- **Cast**: a unary operator (**variable_type**) **explicitly converts the type** the value of an expression to **variable_type**
- To explicitly get the floating-point equivalent of the *integer variable a* you would use a cast and write **(float)a**

```
int i;
char c;
i = c;          /* Implicit promotion */
                /* OK in Java and C */
c = i;          /* Implicit demotion */
                /* Java: Compile time error */
                /* C: OK; truncation */
c = (char)i;    /* Explicit demotion using a cast */
                /* Java: OK; truncation */
                /* C: OK; truncation */
```

Java versus C: Mostly Similar Syntax

```
int x = 42 + (7 * -5);  
double pi = 3.14159;  
char c = 'Q';
```

```
for (int i = 0; i < end; i++) { // variable i is a loop guard  
    if (i % 2 == 0) {  
        x += i;  
    }  
}
```

```
int i; // i initial value is undefined  
...  
if (i) /* is the same as (i != 0) */  
    statement1;  
else  
    statement2;
```

Which statement is executed
after the if statement test?
Depends on what value of i,
is i zero or non-zero

Compiler Warning and unused variable and parameters

- C programming language has many features that when used improperly can lead to runtime issues due to focus on creating code that optimizes performance
 - Example: runtime checks on array boundaries
- gcc besides checking proper language syntax, has the option to include **include heuristic warnings** about potential issues that some consider potential issues in your code
- In CSE30 we require compiling with heuristic checking enabled so you learn to be careful when writing your code, these flags do the checking and requires you to fix them
`gcc -Wall -Wextra`
- As an example, lets look at a couple of warning messages and how to deal with them

Compiler warnings on fall throughs

- When **writing switch statements** in C it is not uncommon to see a case use a **fall through** to the next case below it (this is legal to do in C)
 - **Why do this:** First state does extra steps and then the same steps as the "fall through" state
 - But compilers often (with extra checking flags, using heuristics) decide to flag this as a potential error
 - **The Fix:** use the comment `/* FALL THROUGH */` (a bit of a "hack" 😊)

```
int a = 2;
switch (a) {
case 1:
    printf("1\n");
    break;
case 2:
    printf("2\n");
default:
    printf("default\n");
    break;
}
```

```
int a = 2;
switch (a) {
case 1:
    printf("1\n");
    break;
case 2:
    printf("2\n");
    /* FALL THROUGH */
default:
    printf("default\n");
    break;
}
```

```
% gcc -ggdb -Wall -Wextra switch.c
switch.c: In function 'main':
switch.c:11:9: error: this statement may fall through [-Werror=implicit-fallthrough=]
   11 |         printf("2\n");
      |         ^~~~~~
switch.c:12:5: note: here
   12 |     default:
      |     ^~~~~~
```

```
% gcc -ggdb -Wall -Wextra switch.c
% ./a.out
2
default
%
```

Compiler warnings on unused variables and parameter

- While you are developing a program, you may have functions that you are writing but have not completed the body of the code, but you are compiling it while working on other code
- TEMPORARILY** suppress warning statement use the following for a used variable or parameter: var
`(void) var; // do not submit code to gradescope with this, it will cost you points....`

```
...  
int c = 0;  
...  
state = nextstate(c);  
...
```

```
int nextstate(int c)  
{  
    int j;  
  
    return 0;  
}
```

```
int nextstate(int c)  
{  
    int j;  
  
    (void) c;  
    (void) j;  
    return 0;  
}
```

```
% gcc -c sample.c  
% ls -l  
total 4  
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 45 Mar 28 13:14 sample.c  
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 840 Mar 28 13:17 sample.o  
%  
% gcc -c -Wall -Wextra sample.c  
sample.c: In function 'nextstate':  
sample.c:3:6: error: unused variable 'j' [-Werror=unused-variable]  
    int j;  
      ^  
sample.c:1:19: error: unused parameter 'c' [-Werror=unused-parameter]  
int nextstate(int c)  
                ~~~~^
```

```
% gcc -c -Wall -Wextra sample.c  
% ls -l  
total 4  
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 45 Mar 28 13:18 sample.c  
-rw-r--r-- 1 cs30sp24 ieng6_cs30sp24 840 Mar 28 13:19 sample.o
```

Data types: C Versus Java

Data Types	Java	C
Character	<code>char // 16-bit unicode</code>	<code>char // 8 bits (varies by hardware)</code>
integers	<code>byte // 8 bits</code> <code>short // 16 bits</code> <code>int // 32 bits</code> <code>long // 64 bits</code>	<code>(unsigned, signed) char // see row above</code> <code>(unsigned, signed) short // unspecified</code> <code>(unsigned, signed) int // unspecified</code> <code>(unsigned, signed) long // unspecified</code>
Floating Point	<code>float // 32 bits</code> <code>double // 64 bits</code>	<code>float // unspecified</code> <code>double // unspecified</code>
Logical type	<code>boolean</code>	<code>#include <stdbool.h></code> <code>bool</code> conditional tests that evaluate to 0 are false, true for all other values
Constants	<code>final int MAX = 1000;</code>	<code>// two alternatives to do this</code> <code>#define MAX 1000 // C preprocessor</code> <code>const int MAX = 1000;</code>

C Versus Java

	Java	C
Strings	<code>String s1 = "Hello";</code>	<code>char *s1 = "Hello"; // pointer version</code> <code>char s1[] = "Hello"; // array version</code>
String Concatenation	<code>s1 + s2</code> <code>s1 += s2;</code>	<code>#include <string.h></code> <code>strcat(s1, s2);</code>
Logical ops	<code>&&, , !</code>	<code>&&, , !</code>
Relational ops	<code>==, !=, <, >, <=, >=</code>	<code>==, !=, <, >, <=, >=</code>
Arithmetic ops	<code>+, -, *, /, %, unary -</code>	<code>+, -, *, /, %, unary -</code>
Bitwise ops	<code><<, >>, >>>, &, ^, , ~</code>	<code><<, >>, &, ^, , ~</code>
Assignment ops	<code>=, +=, -=, *=, /=, %=,</code> <code><<=, >>=, >>>=, &=, ^=, =</code>	<code>=, +=, -=, *=, /=, %=,</code> <code><<=, >>=, &=, ^=, =</code>

C Versus Java

	Java	C
Arrays	<pre>int [] a = new int [10]; float [][] b = new float [5][20];</pre>	<pre>int a[10]; float b[5][20];</pre>
Array bounds checking	<pre>// run time checking</pre>	<pre>// no run time checks - speed optimized</pre>
Pointer type	<pre>// Object reference is an // implicit pointer</pre>	<pre>int *p; char *p;</pre>
Record type	<pre>class Mine { int x; float y; }</pre>	<pre>struct Mine { int x; float y; };</pre>

C Versus Java

	Java	C
if, switch, for, do-while, while, continue, break, return	// equivalent	// equivalent
exceptions	throw, try-catch-finally	// no equivalent
labeled break	break somelabel;	// no equivalent
labeled continue	continue somelabel;	// no equivalent
calls: Java method C function	f(x, y, z); someObject.f(x, y, z); SomeClass.f(x, y, z);	f(x, y, z); // other differences, later...

C Versus Java

Note: Sorry for the "poor" code indentation; adjusted to fit into the table

	Java	C
Overall Program Structure	<pre>source file: Hello.java public class Hello { public static void main (String[] args) { System.out.println("hello world!"); } }</pre>	<pre>source file: hello.c #include <stdio.h> #include <stdlib.h> int main(void) { printf("hello world!\n"); return EXIT_SUCCESS; }</pre>
Access a library	<pre>import java.io.File;</pre>	<pre>#include <stdio.h> // may need to specify library at compile time with -llibname</pre>
Building	<pre>% javac Hello.java</pre>	<pre>% gcc -Wall -Wextra hello.c -o hello</pre>
Running (execution)	<pre>% java Hello hello world!</pre>	<pre>% ./hello hello world!</pre>

C Programming Toolchain - Basic Tools

- **gcc**
 - Is a front end for all the tools and by default will turn C source or assembly source into executable programs
- **preprocessor**
 - Insertion into source files during compilation or assembly of files containing macros (expanded), declarations etc.
- **compiler**
 - Translates C programs into hardware dependent assembly language text files
- **assembler**
 - Converts hardware dependent assembly language source files into machine code object files
- **Linker (or link editor)**
 - Combines (links) one or more object files and libraries into executable program files
 - this may include modification of the code to resolve uses with definitions and relocate addresses

C Programming Toolchain: The Source files

- The C development toolchain uses several different file types (indicated by .suffix in the filename)
- **filename.h** public interface *"header or include files" often used as <filename.h> or "filename.h"*
 - **common contents**: public (exported) function and variable declarations, and constants and language macros
 - Processed by **cpp** (the **C pre-processor**) to do inline expansion of the include file contents and insert it into a source file before the compilation starts, enables consistency
- **filename.c**
 - a source text file in **C language source**
 - Processed by **gcc**
- **filename.S**
 - a source text file in **hardware specific assembly language** (programmer created)
 - processed by gcc which calls gas (assembler)
- **filename.s**
 - machine generated by the compiler from a **.c** file
 - processed by gcc which calls gas (assembler)

C Programming Toolchain: The Generated files

- **filename.o** *"relocatable object file"*
 - Compiled from a single source file in a **.c** file or assembled from a single **.s** file into machine code
 - A **.o** file is an incomplete program (not all references to functions or variables are defined) this code will not execute
 - The **.o** and **.c**, **.s**, or **.S** files share the same root name by convention
 - created by gcc calling ld (linkage editor)
- **library.a** *"static library file"*
 - aggregation of individual **.o** files where each can be extracted independently
 - during the process of combining **.o** files into an executable by the **linkage editor**, the files are extracted as needed to **resolve missing definitions**
 - created by **ar**, processed by **ld** (usually invoked via **gcc**)
- **a.out** *"executable program"*
 - Executable program (may be a combination of one or more **.o files and .a files**) that was compiled or assembled into machine code and **all variables and functions are defined**
 - processed by **ld** (usually invoked via **gcc**)

Basic gcc toolchain usage

- Run gcc with flags
 - **-Wall -Wextra**
 - required flag for c programs in cse30
 - output all warning messages
 - **-c**
 - **Optional** flag (lower case)
 - Compile or assemble to object file only do not call **ld** to link
 - creates a **.o** file
 - **-ggdb**
 - **Optional** flag
 - **Compile with debug support** (gdb)
 - generates code that is easier to debug
 - removes many optimizations
 - **-o <filename>**
 - specifies **filename** of executable file
 - **a.out** is the default
 - **-S**
 - upper case **S**, not normally used
 - Compiles to assembly text file and stops
 - creates a **.s** file
- Producing an executable file
 - **gcc -Wall -Wextra mysrc.c**
 - creates an executable file **a.out**
- To use a specific version of C use of one the std= option
 - **gcc -Wall -Wextra -std=c11 mysrc.c**
- Producing an object file with gdb debug support add **-ggdb**
 - **gcc -Wall -Wextra -c -ggdb mysrc.c**
 - creates an object file **mysrc.o**
 - **gcc -Wall -Wextra -c -ggdb mymain.c**
 - creates an object file **mymain.o**
- Linkage step
 - combining a program spread across multiple files
 - **gcc -Wall -Wextra -o myprog mymain.o mysrc.o**
 - creates executable file **myprog**
- Compile and linkage of file(s) in one step
 - **gcc -Wall -Wextra -o myprog mysrc.c mymain.c**
- run the program (refer to cse15l notes)
 - **% ./myprog**

Aside: Remember make from CSE15L?

```
# CSE30SP24 DFA Example

# if you type 'make' without arguments, this is the default
PROG      = noq
all:      $(PROG)

# header files and the associated object files
HEAD      = states.h
SRC       = noq.c states.c
OBJ       = ${SRC:%.c=%.o}

# special libraries
LIB        =
LIBFLAGS   = -L ./ $(LIB)

# select the compiler and flags you can over-ride on command line
# e.g., make DEBUG=
CC         = gcc
DEBUG      = -ggdb
CSTD       =
WARN       = -Wall -Wextra
CDEFS      =
CFLAGS     = -I. $(DEBUG) $(WARN) $(CSTD) $(CDEFS)

$(OBJ):    $(HEAD)

# specify how to compile/assemble the target
$(PROG):   $(OBJ)
           $(CC) $(CFLAGS) $(OBJ) $(LIB) -o $@

# remove binaries
.PHONY: clean clobber
clean:
    rm -f $(OBJ) $(PROG)
```

Programming a Deterministic Finite Automaton - testing

```
$ make
gcc -I. -ggdb -Wall -Wextra -c -o noq.o noq.c
gcc -I. -ggdb -Wall -Wextra -c -o states.o states.c
gcc -I. -ggdb -Wall -Wextra noq.o states.o -o noq
$ ./noq
123"456"789
123789
"123"45"67"
45
"123
456
78"9
9
"test
^d
noq error: Missing end quote "
$ cat in
line1"34"
"line2"line2
line3"
line4
$ ./noq < in > out
noq error: missing end quote "
$ cat out
line1
line2
line3$
```

typed input in red
output in blue