Managing Linux systems with embedded perspective

The Build Process

TX00DG08-300X

C Programs

- Typically C/C++ programs are comprised of Multiple files.
- Each .c/.cpp-file is compiled individually
  - If the source code references external objects (objects from other c/cpp-files) the objects must be declared before they can be used
  - Declarations of functions, types and variables are placed in header files
    - Typically .h files contain no executed code
    - Inclusion of .h files, macro expansions, conditional compilation

Header Files

- Headers are imported to a program by using the #include directive
- # indicates a directive that will be handled by the C-preprocessor
  - Preprocessor replaces include directive with the content from the indicated file
  - Two variants of #include
    - <> = System Header
    - “ ” = User Header

Header Files

- User headers are first searched from the directory where the file to be compiled is
  - Additional search directories may be specified in command line options (or in development environment)
  - If the header to be included is not in the same directory as file to be compiled a path may be specified
    - `#include “compiledDirectory/your_h_file.h”`
  - System headers are located from compiler installation specific path(s)
    - <> = System Header
    - USER HEADERS SHOULD NOT USE <>

C/CPP Build Process

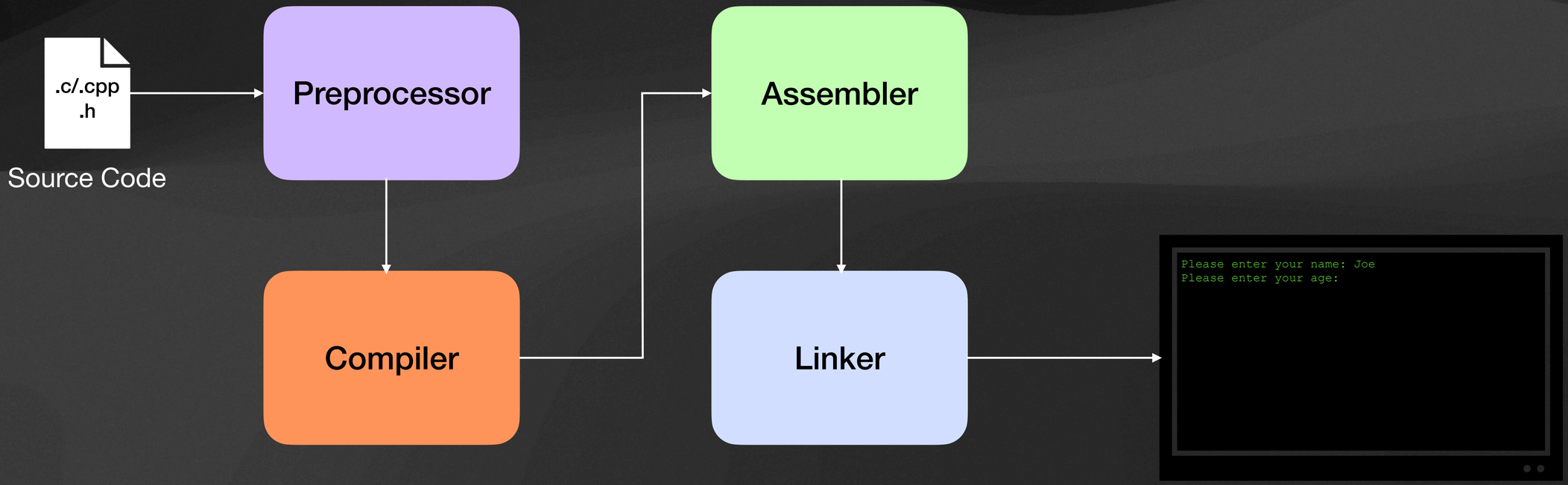
- Typically C/C++ programs are built in the same way
- They go through four steps to buld an executable
  - Preprocessor
    - Macro processor used automatically by the compiler
      - Includes header files
      - Macro expansion
      - Conditional compilation

C/CPP Build Process
Continued

- Compiler
  - Checks syntax
  - Generates Assembly (".s" files)
- Assembly
  - Generates Machine Language (object files ".o" files)
- Linker
  - Connects all of the assembled object files to create an executable

C/CPP Build Process

Continued



Preprocessor

- Preprocessor
  - Handles all directives that begin with #
    - Performs macro expansions
    - Handles conditional compilation
    - Handles comments

Preprocessor

- Typically compiler automatically invokes preprocessor when a C-file is compiled
- It is possible invoke preprocessor manually if you Just want to view the preprocessor steps
- First preprocessor removes comments (converts them to single white space)

Preprocessor
Comment handling

- Two types of comments
  - single line comment prefixed with "//"
  - block comments between "/* */
- Preprocessor deletes all comments and replaces them with a single white space

Preprocessor

Comment example

```c
int main(void) {
    // variable definitions
    int test = 3;
    int val = 2;

    /*
     * A second type of comment
     * for demonstration's purpose
     */

#if 0
    val += test;
#endif
#if 1
    val *= test;
#endif
    return 0;
}
```

```c
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2


int main(void) {

    int test = 3;
    int val = 2;
# 17 "test.c"
    val *= test;

    return 0;
}
```

Preprocessor

Include Directive

- Used for including User or system headers with the "#include" directive
  - User headers included with "" (#include"path/to/header.h")
    - If no path is given the compiler will search in the same directory as the C files
  - System headers included with <> (#include<someheader.h>)
    - The compiler searches known system directories for these headers, unless otherwise directed elsewhere with the `-I` flag
  - The compiler essentially replaces the include directive with the content from the specified header file

Preprocessor

Include Directive

File: **test.c**

```c
#include "test.h"

int main(void) {
    // variable definitions
    int test = 3;
    int val = 2;

    /*
     * A second type of comment
     * for demonstration's purpose
     */

#if 0
    val += test;
#endif
#if 1
    val *= test;
#endif
    return 0;
}
```

File: **test1.c**

```c
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2

int main(void) {
    int test = 3;
    int val = 2;
# 17 "test.c"
    val *= test;

    return 0;
}
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
# 1 "./test.h" 1
int testFunc(int a, float b);
# 2 "test.c" 2

int main(void) {

    int test = 3;
    int val = 2;
# 17 "test.c"
    val *= test;

    return 0;
}
```

Include guards

Prevent including the same .h file more than once

- Constants can be tested in conjuntion with `#if`
- defined(<constant name>) evaluates to non-zero if this constant is defined (it needs not to have a value) and zero if it is not defined
- defined() is so common that there are two shorthand notations for it
  - `#ifdef <name> = #if defined(<name>)`
  - `#ifndef <name> = #if !defined(<name>)`

Include Guards

cont

- Prevent a header from being included more than once
- Place ”guards” in the beginning and end

```
#ifndef _FILE_NAME_H_

#define _FILE_NAME_H_

/* code */

#endif
```

Include Guards

cont

File: **test.c**

```c
#include "test.h"

#include "test1.h"

#define TEST 10

int main(void) {
    // variable definitions
    int test = TEST;
    int val = 2;

    /*
     * A second type of comment
     * for demonstration's purpose
     */

#if 0
    val += test;
#else
    val *= test;
#endif
    return 0;
}
```

File: **test.h**

```c
#ifndef _TEST_H_
#define _TEST_H_

int testFunc(int a, float b);

#endif
```

File: **test1.h**

```c
#include "test.h"
```

File: **test1.c**

```c
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
# 1 "./test.h" 1


int testFunc(int a, float b);
# 2 "test.c" 2

# 1 "./test1.h" 1
# 4 "test.c" 2



int main(void) {

    int test = 10;
    int val = 2;
# 20 "test.c"
    val *= test;

    return 0;
}
```

Preprocessor

Macro expansion

- Preprocessor replaces macros with their defined values in all source files

Preprocessor

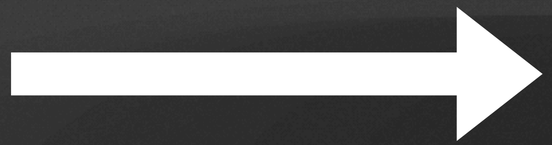
Macro expansion

File: **test.h**

```
1  #ifndef _TEST_H_
2  #define _TEST_H_
3
4  #define PI 3.14159
5  int testFunc(int a, float b);
6
7  #endif
```

File: **test.c**

```
1  #include "test.h"
2
3  #include "test1.h"
4
5  #define RADIUS 10
6
7  #define CIRC 2 * PI* RADIUS
8  int main(void) {
9      // variable definitions
10     int test = 5;
11     int val = 2;
12
13     /* A second type of comment for demonst
14
15 #if 1
16     CIRC;
17 #else
18     val *= test;
19 #endif
20     return 0;
21 }
```



File: **test1.c**

```
1  # 1 "test.c"
2  # 1 "<built-in>" 1
3  # 1 "<built-in>" 3
4  # 418 "<built-in>" 3
5  # 1 "<command line>" 1
6  # 1 "<built-in>" 2
7  # 1 "test.c" 2
8  # 1 "./test.h" 1
9
10
11
12
13 int testFunc(int a, float b);
14 # 2 "test.c" 2
15
16 # 1 "./test1.h" 1
17 # 4 "test.c" 2
18
19
20
21
22 int main(void) {
23
24     int test = 5;
25     int val = 2;
26
27
28
29
30     2 * 3.14159* 10;
31
32
33
34     return 0;
35 }
```

Constants(Macros)

- Constants are defined with #define directive
  - `#define <name of constant> <value of constant>`
- Any occurrence of constant will be replaced with the value that was given in the definition
- For example
  - ```
    #define PI 3.14
    ...
    i = PI * radius;  =>  i = 3.14 * radius;
    ```
    ```

Constants(Macros)

- Value that is given in the definition does not have to be a single literal
  - For example

```
#define PI_PLUS_ONE (3.14 + 1)
```

    - Parentheses are added to guarantee correct order of evaluation
    - Consider following:

```
#define PI_PLUS_ONE (3.14 + 1)

#define PI_PLUS_ONX 3.14 + 1

i = PI_PLUS_ONE * 4; → i = (3.14 + 1) * 4;

i = PI_PLUS_ONX * 5; → i = 3.14 + 1 * 5;
```

Constants(Macros)

- Value can contain other constants
  - For example

```
#define PI 3.14

#define PI_PLUS_ONE (PI + 1)

i = PI_PLUS_ONE * 6; → i = (3.14 + 1) * 6;
```

- It is possible to define constants that have no value

```
#define TEST
```

- This defines a constant called TEST without giving it a value
- Defining constant without values are ofter used in conjunction with conditional compilation

Predefined macros

- String literals
- `__func__`
  - Name of the function
- `__DATE__`
  - Compilation date
- `__FILE__`
  - Name of the source file
- `__LINE__`
  - Current line number
- `__TIME__`
  - Compilation time
- Integer constants
  - `__STDC__`
  - `__STDC_HOSTED__`
  - `__STDC_VERSION__`

Macros

- Macros behave in similar fashion as constants
- Macros have one or more parameters
  - Parameters are type neutral (due to find/replace behavior)
    - There is no type checking!
- Macro definition
  - `#define MACRO_NAME( arg1, arg2, ...) <code to expand to>`
  - Arguments are first replaced and then the result in placed where the macro is invoked
- For example
  - `#define SQUARE(x) x * x`
  - `i = SQUARE(p); → p * p;`
  - `i = SQUARE(7); → 7 * 7;`
  - `i = SQUARE(3 + 5); → i = 3 + 5 * 3 + 5;`
- `Macro is not a function call`
  - `i = SQUARE(3 + 5); → i = 3 + 5 * 3 + 5;`

Variable number arguments

- C99 allows defining macros with an ellipsis (...) at then end of the parameter list to represent optional arguments. In the replacement text, the identifier __VA_ARGS__ represents the group of optional arguments
- For example:
  - `#define printlog(...) fprintf(fp_syslog, __VA_ARGS__) printlog(”%s: count = %d\n”, __func__, count);`

Conditional Compilation

- Preprocessor provides a wide range of directives for conditional compilation
  - `#if, #elif, #else, #ifdef, #ifndef` and `#endif`
- The key directive #if (and its variants)
- Preprocessor evaluates the value after `#if` directive
  - If true (= non zero) then all code until `#else, #elif` or `#endif` is processed normally
  - If not true (= zero) then code is removed

Conditional Compilation

File: **test.c**

```c
#include "test.h"

#include "test1.h"

#define RADIUS 10

#define CIRC 2 * PI* RADIUS
int main(void) {
    // variable definitions
    int test = 5;
    int val = 2;

    /* A second type of comment for demonst

#if 1
    CIRC;
#else
    val *= test;
#endif
    return 0;
}
```

File: **test1.c**

```c
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
# 1 "./test.h" 1




int testFunc(int a, float b);
# 2 "test.c" 2

# 1 "./test1.h" 1
# 4 "test.c" 2




int main(void) {

    int test = 5;
    int val = 2;




    2 * 3.14159* 10;



    return 0;
}
```

Conditional Compilation

File: **test.c**

```c
#include "test.h"

#include "test1.h"

#define RADIUS 10

#define CIRC 2 * PI* RADIUS
int main(void) {
    // variable definitions
    int test = 5;
    int val = 2;

    /* A second type of comment for demonst

#if 0
    CIRC;
#else
    val *= test;
#endif
    return 0;
}
```

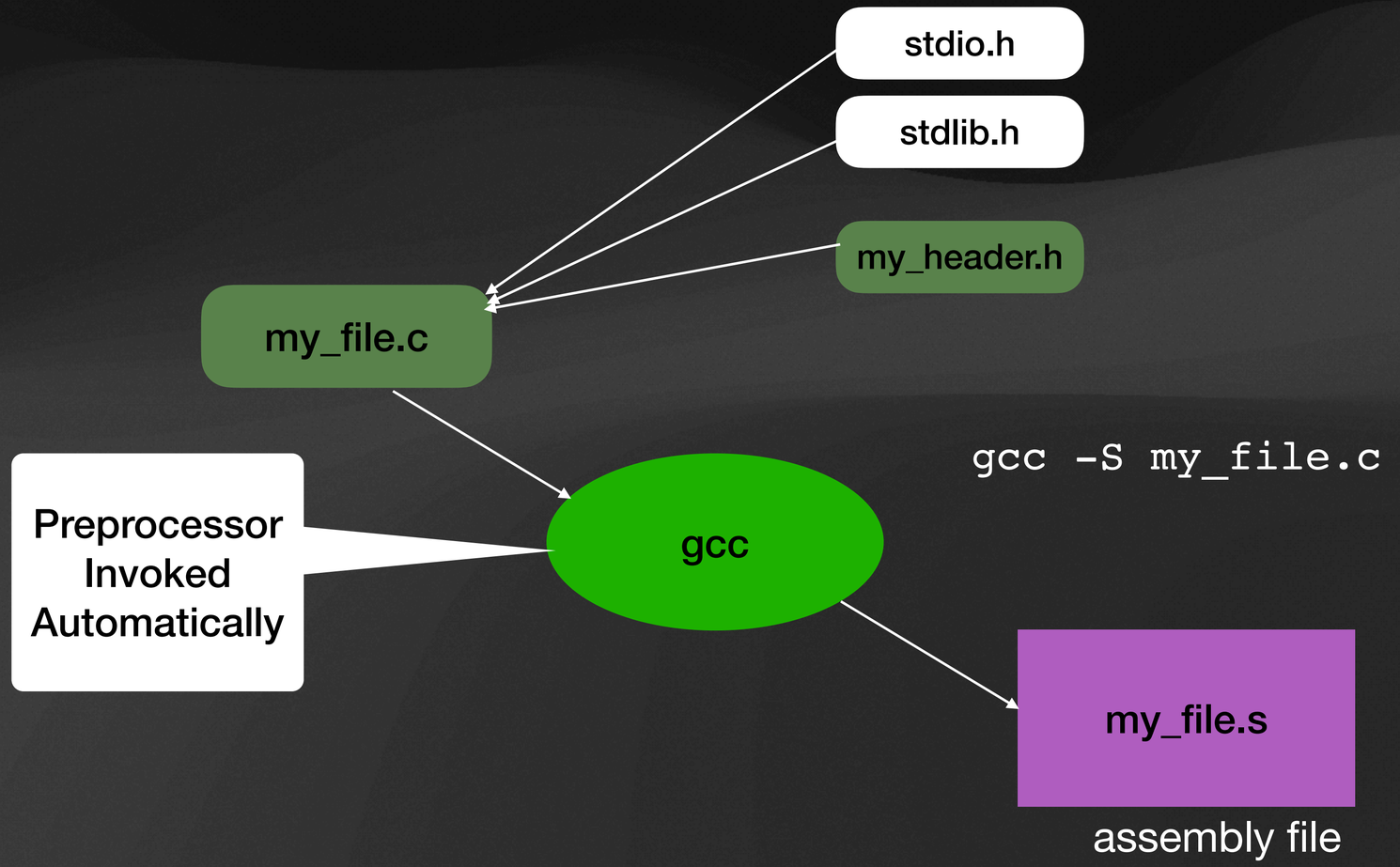
File: **test1.c**

```c
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 418 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
# 1 "./test.h" 1




int testFunc(int a, float b);
# 2 "test.c" 2

# 1 "./test1.h" 1
# 4 "test.c" 2




int main(void) {

    int test = 5;
    int val = 2;









    val *= test;

    return 0;
}
```

Compiler

- Essentially a translator
  - Reads input files of a known programming language
  - Analyzes the code for errors
  - Translates it into a new format suited to a specific platform/architecture
    - There will be an intermediate language between the source code and the final machine code
  - There are additional things that the compiler does, such as optimization

Compilation



Compilation

File: **test.c**

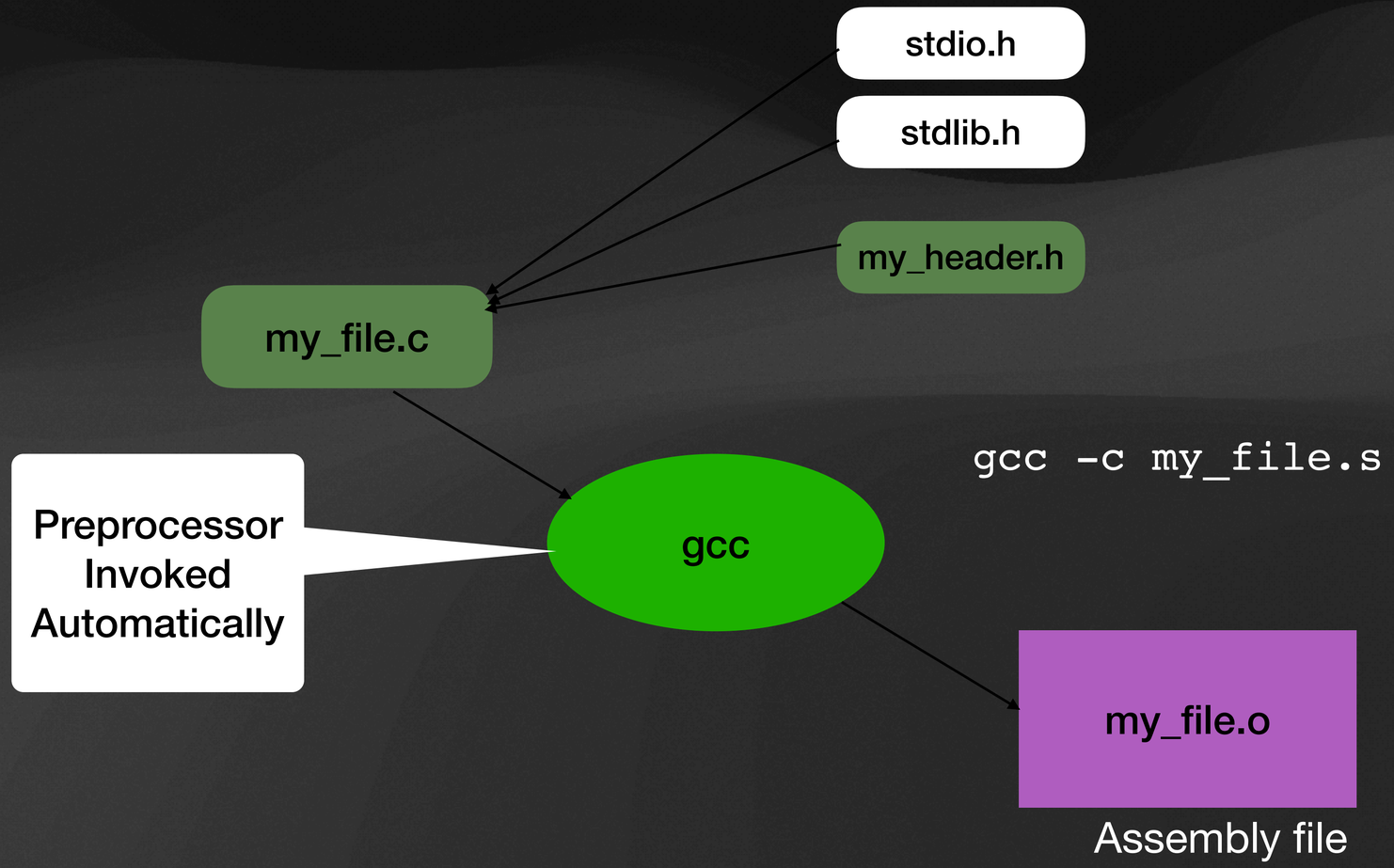
```c
#include "test.h"

#include "test1.h"

#define RADIUS 10

#define CIRC 2 * PI* RADIUS
int main(void) {
    // variable definitions
    int test = 5;
    int val = 2;

    /* A second type of comment for demonstration's purpose */

#if 0
    CIRC;
#else
    val *= test;
#endif
    return 0;
}
```

```
 ~/Doc/M/t/2024-2025/Period_1/C_CPP/examples/Week1                    at 19:23:14
❯
```

File: **test.s**

```asm
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 14, 0 sdk_version 14, 4
    .globl  _main                           ; -- Begin function main
    .p2align    2
_main:                                  ; @main
    .cfi_startproc
; %bb.0:
    sub sp, sp, #16
    .cfi_def_cfa_offset 16
    mov w0, #0
    str wzr, [sp, #12]
    mov w8, #5
    str w8, [sp, #8]
    mov w8, #2
    str w8, [sp, #4]
    ldr w9, [sp, #8]
    ldr w8, [sp, #4]
    mul w8, w8, w9
    str w8, [sp, #4]
    add sp, sp, #16
    ret
    .cfi_endproc
                                        ; -- End function
.subsections_via_symbols
```

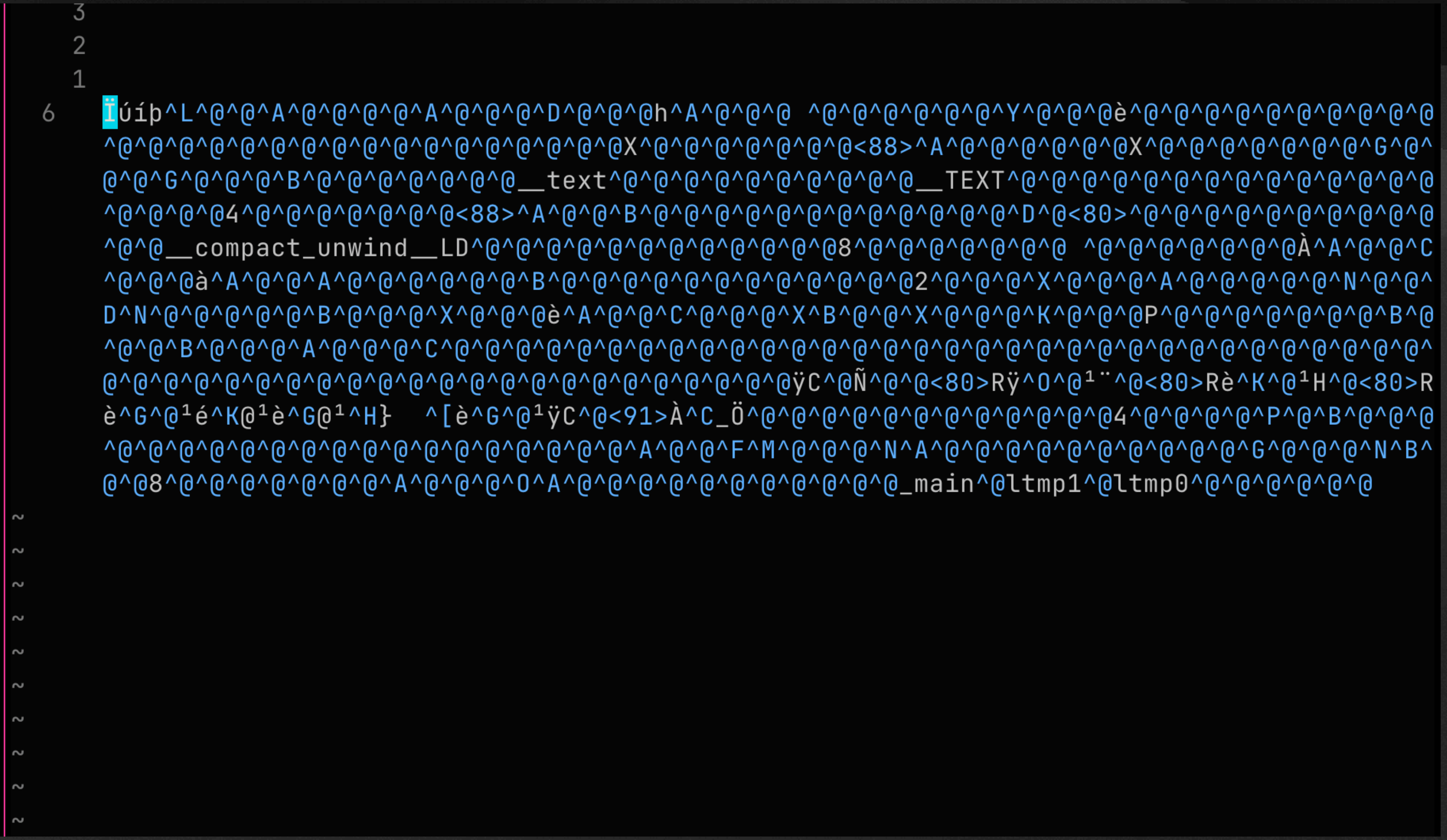
Compilation



Compilation

File: test.c

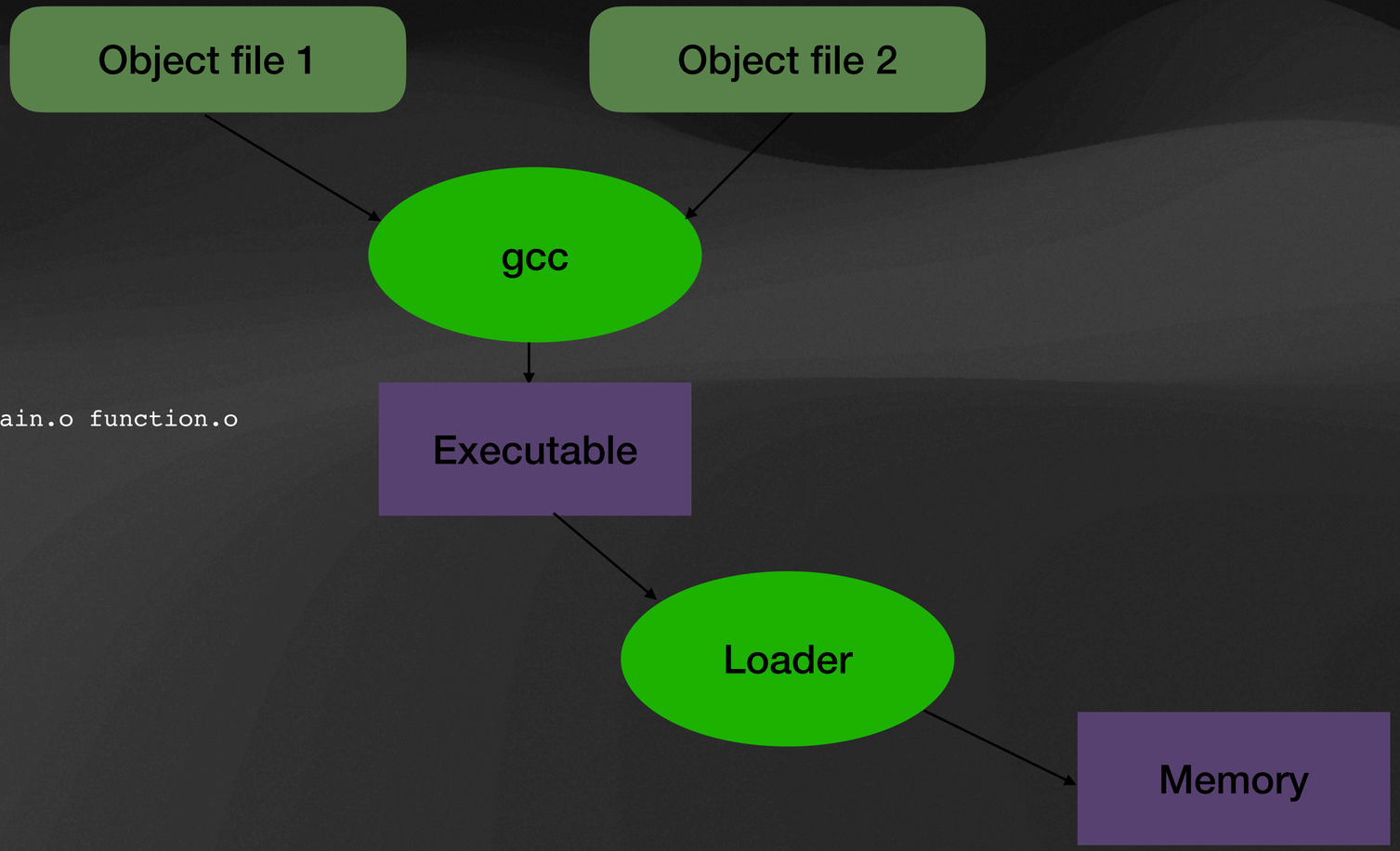
```c
#include "test.h"

#include "test1.h"

#define RADIUS 10

#define CIRC 2 * PI* RADIUS
int main(void) {
    // variable definitions
    int test = 5;
    int val = 2;

    /* A second type of comment for demonstration's purpose */

#if 0
    CIRC;
#else
    val *= test;
#endif
    return 0;
}
```



Linking

- A linker is a program that makes an executable program from one or more object files
- Modern IDEs abstract most of the details of compilation and linking from you.
- Pressing the build button will execute the compiler and linker automatically.

Linking




```
gcc -o my_prog main.o function.o
```

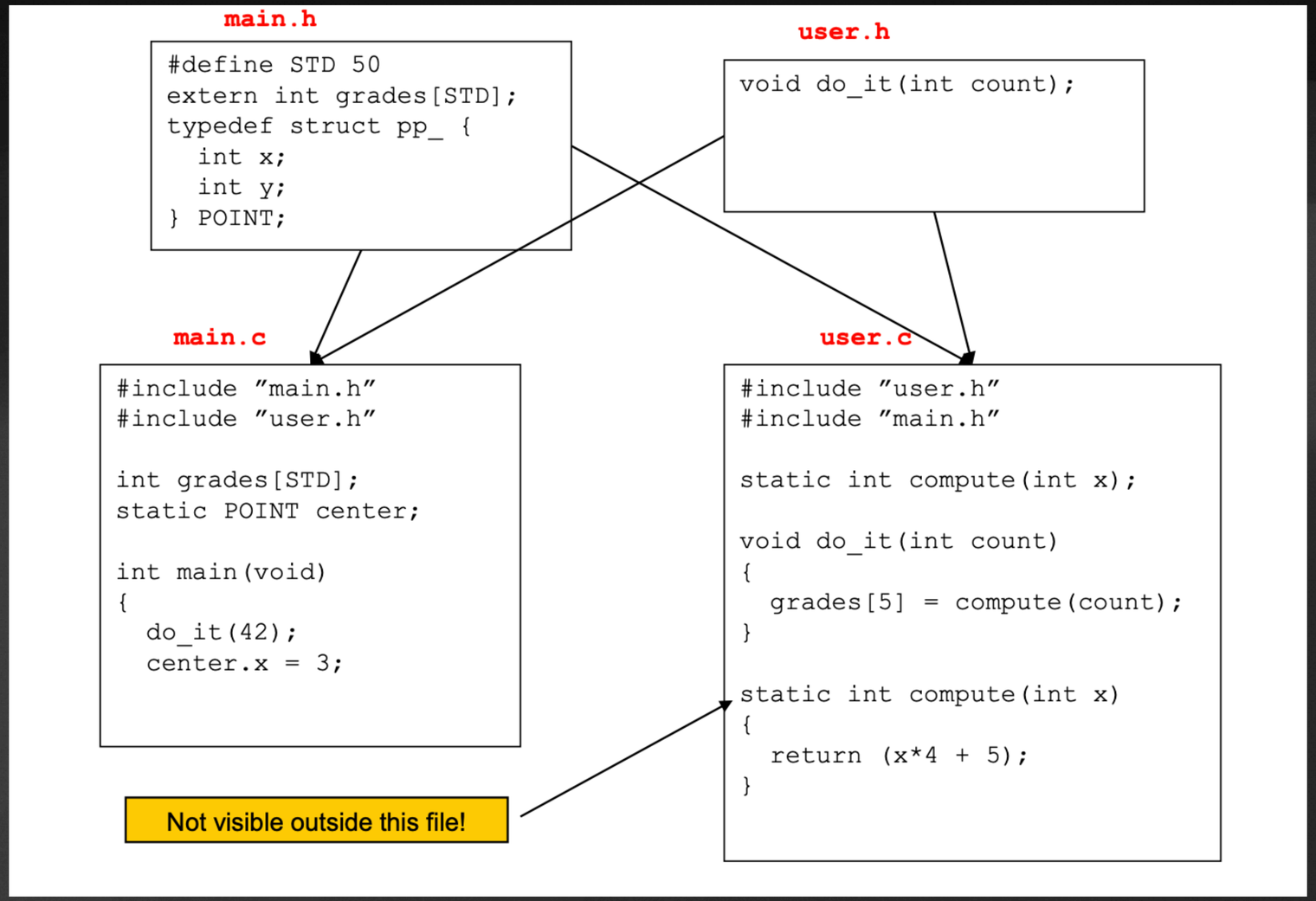
Variable scope

- Global Variables
  - Defined outside of functions
  - Visible across files
  - Globals must be declared to indicate that a global variable was defined in another file
  - Global variables can be declared, by using the extern keyword
  - Globals defined with the static keyword can only be seen in the file where it is defined
- Avoid using global variables >>> why?

Function scope

- Global by default
- Limit scope with `static` keyword
  - If a function declaration and definition is preceded by keyword static then the defined function is visible only within the file where it is defined
  - It is good programming practice to declare all functions in the beginning of the file (or you can import declarations with #include)
  - Declare as many functions static as possible to prevent namespace pollution

Scope


main.h
#define STD 50
extern int grades[STD];
typedef struct pp_ {
int x;
int y;
} POINT;
user.h
void do_it(int count);
main.c
#include ”main.h”
#include ”user.h”
int grades[STD];
static POINT center;
int main(void)
{
do_it(42);
center.x = 3;
user.c
#include ”user.h”
#include ”main.h”
static int compute(int x);
void do_it(int count)
{
grades[5] = compute(count);
}
static int compute(int x)
{
return (x*4 + 5);
}
Not visible outside this file!

Scope

Example

Function(s) for reading user input

Functions for array handling

```
#include "read.h"
#include "operations.h"

int main(void)
{
    STUDENT group_a[STC] =
    {
        { "John Lydon", 24 },
        { "Ian Kilmister", 21 },
        { "Antti Hulkko", 22 },
        { "", 0 }
    };
    int x;

    return 0;
}
```

```
do
{
    printf("Select:\n");
    printf("1 - Add\n");
    printf("2 - Remove\n");
    printf("3 - Print\n");
    printf("9 - exit\n");
    x = read_int("Select operation: ");
    switch (x)
    {
    case 1:
        add_student(group_a);
        break;
    case 2:
        remove_student(group_a);
        break;
    case 3:
        print_students(group_a);
        break;
    case 9:
        printf("Goodbye!\n");
        break;
    default:
        printf("Illegal selection\n");
        break;
    }
}
while (x != 9);
```