

# 601.220 Intermediate Programming

Separate compilation

# Outline

- Header files and separate compilation

# Separate source files

- Big software projects are typically split among multiple files
- Helper functions might be separated from main, some code grouped together as a library of functions which accomplish related tasks
- Different developers may create/edit/test different pieces

# Header files

- But how do we allow different files used within one program to communicate?
  - Must specify definitions and declarations that different files need to use
  - Typically, we use header files (`.h` files) to group together declarations, then `#include` them into appropriate files
  - A separate `.c` source file will contain definitions for those functions declared in a `.h` header file. Typically, functions defined in `file.c` are declared in a function named `file.h`

# Header files in C

- When the preprocessor sees a `#include` directive, it inserts the contents of the specified file at that location in the code
  - Generally, the included file contains *declarations* of functions that are used in the code
  - Note that there are two ways to include files
    - `#include <stdio.h>`
    - `#include "myHeader.h"`

# Using header files

```
// functions.c:  
#include "functions.h" //including my own header file; use " "  
  
float func1 (int x, float y) {  
    return x+y;  
}  
  
int func2 (int a) {  
    return 2*a;  
}  
  
// functions.h:  
float func1 (int x, float y);  
int func2 (int a);
```

# Using header files

```
// mainFile.c:
#include <stdio.h>
#include "functions.h"

int main() {
    printf("%.2f %d", func1(2,3.0), func2(7));
    return 0;
}

$ gcc -std=c99 -pedantic -Wall -Wextra mainFile.c functions.c
$ ./a.out
5.00 14
```

# Using header files

```
// mainFile.c:  
#include <stdio.h>  
//#include "functions.h" //leaving this out!
```

```
int main() {  
    printf("%.2f %d", func1(2,3.0), func2(7));  
    return 0;  
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra mainFile.c functions.c
```

```
mainFile.c: In function 'main':
```

```
mainFile.c:5:22: warning: implicit declaration of function 'func1' [-Wimplicit-
```

```
5 |     printf("%.2f %d", func1(2,3.0), func2(7));  
  |                                     ^~~~~~
```

```
mainFile.c:5:36: warning: implicit declaration of function 'func2' [-Wimplicit-
```

```
5 |     printf("%.2f %d", func1(2,3.0), func2(7));  
  |                                     ^~~~~~
```

```
mainFile.c:5:15: warning: format '%f' expects argument of type 'double', but ar
```

```
5 |     printf("%.2f %d", func1(2,3.0), func2(7));  
  |           ~~~~^      ~~~~~~  
  |           |         |  
  |           double int  
  |           %.2d
```



# Checkpoint Question!

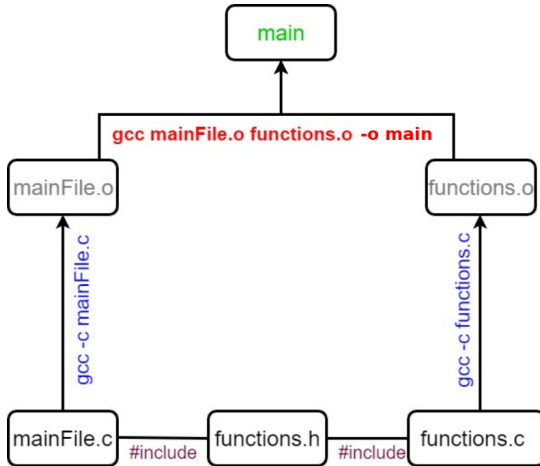
Which statement is wrong?

- A. We use double quotation marks with `#include` to include our own header files, e.g. `#include "myHeader.h"`
- B. Generally, a header file only contains the declarations not the actual implementations
- C. The compiler needs full function definition in order to produce the object file(s)
- D. When declaring a function, it is optional to write the names of the parameters
- E. None of the above

# Compiling and Linking

- Until now, we've used one `gcc` command to perform compilation and linking steps for us
  - *compiling* translates source files (`.c` files) into intermediate object files (`.o` files)
  - *linking* combines `.o` files into one executable file called `a.out` (Recall that we can optionally specify the executable name with `-o` flag)

# Compiling and Linking



# Using header files

```
// mainFile.c:
#include <stdio.h>
#include "functions.h" //put header file back in

int main() {
    printf("Calling functions..."); //added this line
    printf("%.2f %d", func1(2,3.0), func2(7));
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra mainFile.c functions.c
$ ./a.out
Calling functions...5.00 14
```

- The gcc command above recompiled functions.c, even though nothing changed in it

# Compiling and Linking

- But a change to one source file doesn't always necessitate re-compiling all source files! Just re-compile what changed (and anything that depends on it), and then re-link
  - To separate compiling from linking, we can use the `-c` flag with `gcc` to indicate that we want to compile a source file to create an object file, and then separately run `gcc` on the resulting object files once all have compiled.
  - `gcc -c` command generates an object file with `.o` extension

# Compiling and Linking Separately

- Compile source functions.c to create object file functions.o:
  - `gcc -std=c99 -pedantic -Wall -Wextra -c functions.c`
- Compile source mainFile.c to create object file mainFile.o:
  - `gcc -std=c99 -pedantic -Wall -Wextra -c mainFile.c`
- Link two object files to create executable named main:
  - `gcc -o main mainFile.o functions.o`
- Run the resulting executable file:
  - `./main`

# Using header files

```
// mainFile.c:
#include <stdio.h>
#include "functions.h" //put header file back in

int main() {
    printf("Calling functions..."); //added this line
    printf("%.2f %d", func1(2,3.0), func2(7));
    return 0;
}

$ gcc -std=c99 -pedantic -Wall -Wextra -c functions.c
$ gcc -std=c99 -pedantic -Wall -Wextra -c mainFile.c
$ gcc -o main mainFile.o functions.o
$ ./main
Calling functions...5.00 14
```

# Using header files

```
// mainFile.c:
#include <stdio.h>
#include "functions.h" //put header file back in

int main() {
    //removed the line that was here, so now re-compile this
    printf("%.2f %d", func1(2,3.0), func2(7));
    return 0;
}
```

```
$ gcc -std=c99 -pedantic -Wall -Wextra -c mainFile.c
$ gcc -o main mainFile.o functions.o
$ ./main
5.00 14
```

- No need to recompile the functions.c code.