

Header files

Adapted from materials by Dr. Carrier



A scenario...

Imagine we've written an incredible function or struct

A scenario...

Imagine we've written an incredible function or struct

```
void PrintArray(int* arr, int len){  
    printf("[");  
    for(int i = 0; i < len; i++){  
        if(i != 0) printf(" ");  
        printf("%d", *(arr + i));  
        if(i != len-1) printf(",");  
    }  
    printf("]\n");  
}
```

A scenario...

Imagine we've written an incredible function or struct

```
void PrintArray(int* arr, int len){  
    printf("[");  
    for(int i = 0; i < len; i++){  
        if(i != 0) printf(" ");  
        printf("%d", *(arr + i));  
        if(i != len-1) printf(",");  
    }  
    printf("]\n");  
}
```

How can we reuse this code in different files?

Introducing: header files

Header files contain C code, filenames end in .h

Introducing: header files

Header files contain C code, filenames end in .h

Where have we seen these before?

Introducing: header files

Header files contain C code, filenames end in .h

Where have we seen these before?

The files we've been including, example:

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

Introducing: header files

Header files contain C code, filenames end in .h

Where have we seen these before?

The files we've been including, example:

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

For system headers, we use

```
#include <file.h>
```


Introducing: header files

Header files contain C code, filenames end in .h

Where have we seen these before?

The files we've been including, example:

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

For system headers, we use

```
#include <file.h>
```

For local headers that we write:

```
#include "file.h"
```

What goes in a header file

Traditionally, header files contain structs, variables, and function *prototypes*

What goes in a header file

Traditionally, header files contain structs, variables, and function *prototypes*

(You can put function definitions in headers, there are tradeoffs)

Example header file (print.h)

```
#include <stdio.h>

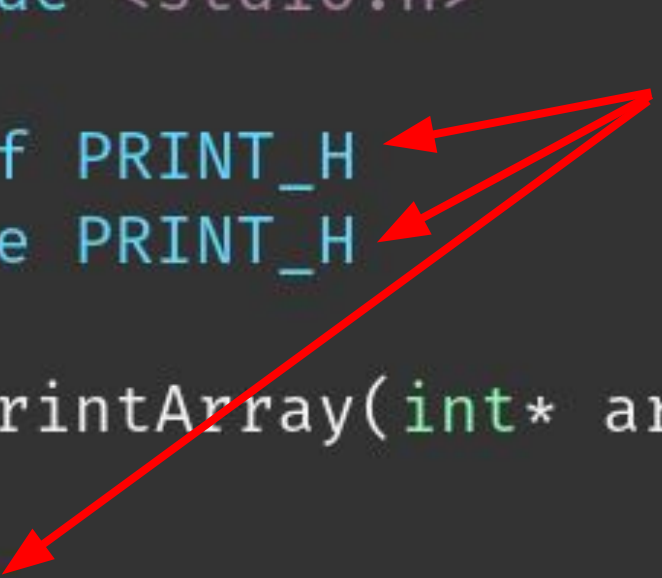
#ifndef PRINT_H
#define PRINT_H

void PrintArray(int* arr, int len);

#endif
```

Example header file (print.h)

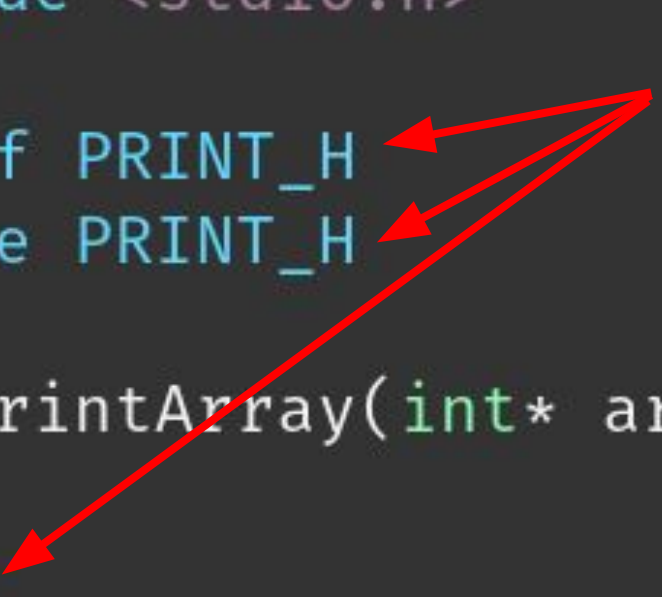
```
#include <stdio.h>
                                ???
#ifndef PRINT_H
#define PRINT_H
void PrintArray(int* arr, int len);
#endif
```



The diagram illustrates the concept of a header guard. Three red arrows originate from the red text '???' and point to the preprocessor directives `#ifndef PRINT_H`, `#define PRINT_H`, and `#endif` in the code snippet, highlighting their role in preventing multiple inclusions of the header file.

Example header file (print.h)

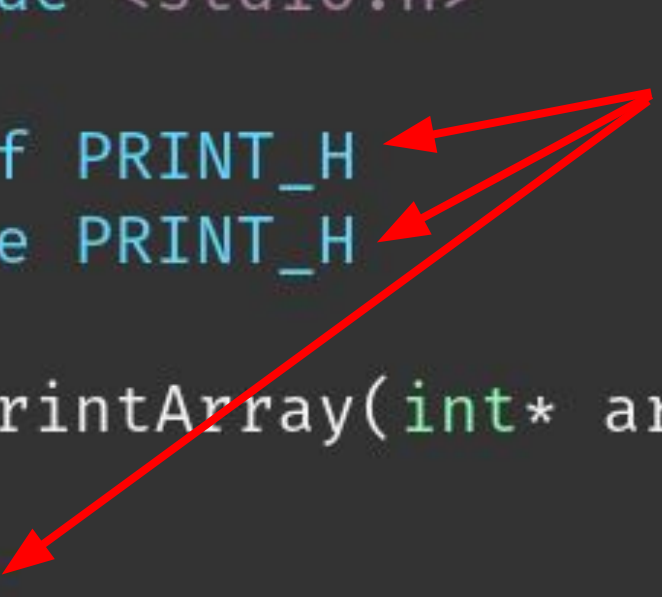
```
#include <stdio.h>
???
#ifndef PRINT_H
#define PRINT_H
void PrintArray(int* arr, int len);
#endif
```

A diagram with three red arrows originating from the text '???' and pointing to the lines '#ifndef PRINT_H', '#define PRINT_H', and '#endif' in the code block above.

This extra stuff is a header guard

Example header file (print.h)

```
#include <stdio.h>
???
#ifndef PRINT_H
#define PRINT_H
void PrintArray(int* arr, int len);
#endif
```

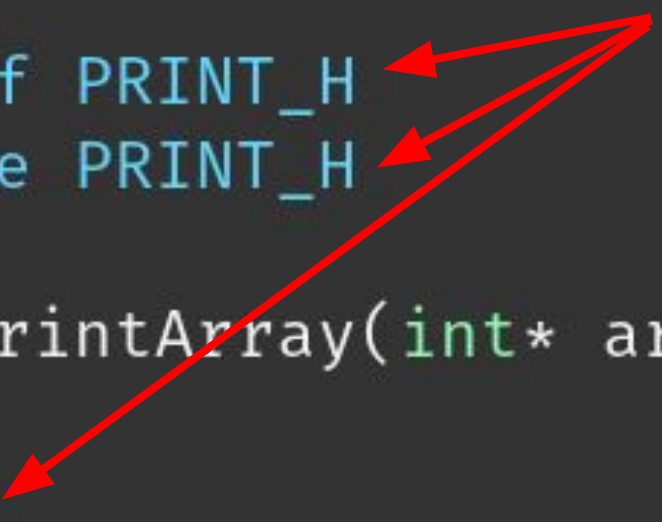
A diagram with three red arrows originating from the text '???' and pointing to the lines '#ifndef PRINT_H', '#define PRINT_H', and '#endif' in the code block above.

This extra stuff is a header guard

`#ifndef` checks to see if macro (`PRINT_H` here) is defined, if so, it cuts out all code between it and `#endif`

Example header file (print.h)

```
#include <stdio.h>
                                ???
#ifndef PRINT_H
#define PRINT_H
void PrintArray(int* arr, int len);
#endif
```



This extra stuff is a header guard

`#ifndef` checks to see if macro (`PRINT_H` here) is defined, if so, it cuts out all code between it and `#endif`

Because we define the macro inside this if, we ensure we have, at most, one copy of this code!

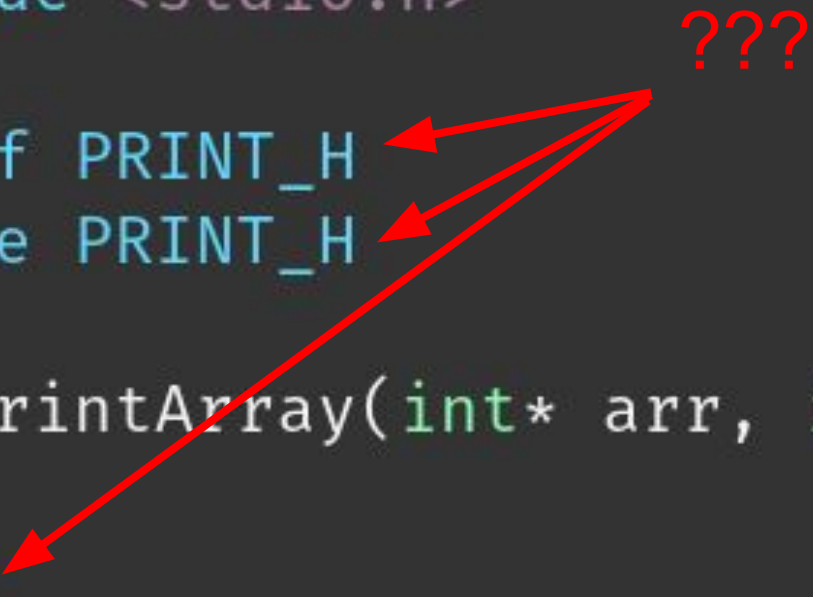
Example header file (print.h)

```
#include <stdio.h>

#ifdef PRINT_H
#define PRINT_H

void PrintArray(int* arr, int len);

#endif
```

A diagram illustrating the header guard logic. Three red arrows originate from the text '???' and point to the lines `#ifdef PRINT_H`, `#define PRINT_H`, and `#endif` in the code block above. This indicates that the code between `#ifdef` and `#endif` is only included if `PRINT_H` is not already defined.

This extra stuff is a header guard

`#ifndef` checks to see if macro (`PRINT_H` here) is defined, if so, it cuts out all code between it and `#endif`

Because we define the macro inside this if, we ensure we have, at most, one copy of this code!

`#pragma once` is a modern alternative

Where does the function definition go?

Functions should be in a separate .c file with the same name

Where does the function definition go?

Functions should be in a separate .c file with the same name

```
#include "print.h"

void PrintArray(int* arr, int len){
    printf("[");
    for(int i = 0; i < len; i++){
        if(i != 0) printf(" ");
        printf("%d", *(arr + i));
        if(i != len-1) printf(",");
    }
    printf("]\n");
}
```

(print.c)

Using our header file

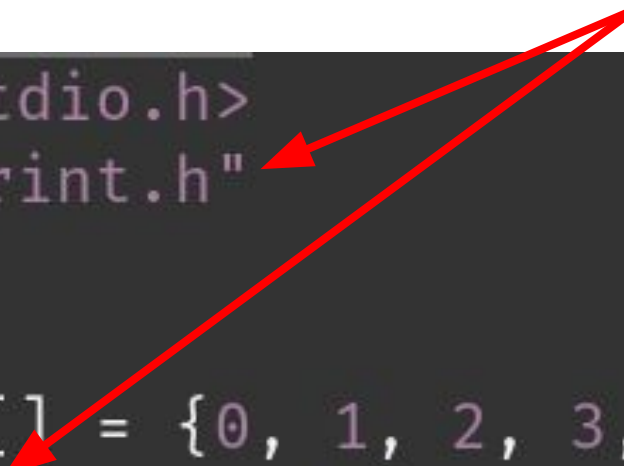
```
#include <stdio.h>
#include "print.h"

int main(){
    int arr_1[] = {0, 1, 2, 3, 4, 5};
    PrintArray(arr_1, 6);
    int arr_2[3];
    arr_2[0] = 1000000;
    arr_2[1] = -57;
    arr_2[2] = 0;
    PrintArray(arr_2, 3);
    return 0;
}
```

Using our header file

```
#include <stdio.h>
#include "print.h"

int main(){
    int arr_1[] = {0, 1, 2, 3, 4, 5};
    PrintArray(arr_1, 6);
    int arr_2[3];
    arr_2[0] = 1000000;
    arr_2[1] = -57;
    arr_2[2] = 0;
    PrintArray(arr_2, 3);
    return 0;
}
```

Two red arrows originate from a single point centered above the space between the two include statements. One arrow points diagonally down and to the left towards the second include statement, "print.h". The other arrow points diagonally down and to the left towards the opening square bracket of the array declaration 'int arr_1[]'.

Compiling

Our program needs to know about both .c files

To compile:

```
gcc program.c print.c
```

Compiling

Our program needs to know about both .c files

To compile:

```
gcc program.c print.c
```

What if we don't want to compile all files each time?
E.g., maybe print.c rarely changes, no reason to recompile.

We can compile them separately using object files:

Compiling

Our program needs to know about both .c files

To compile:

```
gcc program.c print.c
```

What if we don't want to compile all files each time?
E.g., maybe print.c rarely changes, no reason to recompile.

We can compile them separately using object files:

```
gcc -c print.c (creates print.o)
```

```
gcc program.c print.o
```


Compiling

Our program needs to know about both .c files

To compile:

```
gcc program.c print.c
```

What if we don't want to compile all files each time?
E.g., maybe print.c rarely changes, no reason to recompile.

We can compile them separately using object files:

```
gcc -c print.c (creates print.o)
```

```
gcc program.c print.o
```

We are linking files together!
(ever had a linker error?)

Where do includes come from?

A local include like: `#include "file.h"` will first look in current directory

Where do includes come from?

A local include like: `#include "file.h"` will first look in current directory

System also has some go-to directories that can be checked.

Where do includes come from?

A local include like: `#include "file.h"` will first look in current directory

System also has some go-to directories that can be checked.

We can also specify paths to check for files in our gcc call using `-I`

Where do includes come from?

A local include like: `#include "file.h"` will first look in current directory

System also has some go-to directories that can be checked.

We can also specify paths to check for files in our gcc call using `-I`

Example:

```
gcc -Iother_proj/headers
```

More on preprocessor directives

Remember: these operate on **text**, they do not consider code context

More on preprocessor directives

Remember: these operate on **text**, they do not consider code context

List (part 1):

`#include <header.h>` -> paste contents of header.h

`#define NAME X` -> Replace instances of NAME with X

`#define NAME(a) printf("a");` -> Can also take arguments

`#undef NAME` -> Undefines NAME

More on preprocessor directives

Remember: these operate on **text**, they do not consider code context

List (part 2):

`#if X` -> Includes following code if X is not 0

`#elseif` and `#else` -> Go with if

`#endif` -> Ends if / elseif / else blocks

`#ifdef X` -> Like if; includes code if X is defined

`#ifndef X` -> Includes if X is NOT defined