More on the C Preprocessor

For use in CSE6010 only

Not for distribution

The C Preprocessor

- The C preprocessor looks at your program before it is compiled
- It follows the preprocessor directives to
  - Replace symbolic abbreviations
  - Include other files
  - Select which code the compiler sees
- Essentially, it takes some text and converts it to other text!
- We've already seen a few examples
  - Basic #define
  - #include

A few other important directives

- Conditional compilation:
  - #include guards with #ifndef … #endif
  - #ifdef, #else, #endif
- Predefined macros that may be useful

#include guards

- C can have problems if the same macro is included more than once.
- This can happen if header1.h includes header2.h, and both header1.h and header2.h are included in main.c.
  - main.c will have two copies of whatever is in header2.h.
  - C will complain or have problems (e.g., give error for multiple definitions of a struct, ignore definitions after the first header included, etc.).
- To avoid, use an include guard/header guard: a conditional compilation directive.
  #ifndef (if not defined) [skip all that follows until #endif is reached]

Example

header-1.h

```
typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);
```

header-2.h

```
#include "header-1.h"

int myFunction2(MyStruct *value);
```

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

- Problem: MyStruct is defined twice.
- Compilation error will result.
- May be possible to prevent the error by being very careful about what is included.
- Easier: use include guards to prevent the same file from being included more than once.

https://riptutorial.com/c/example/7662/header-include-guards

Implementing #include guards

header-1.h

```c
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif
```

header-2.h

```c
#ifndef HEADER_2_H
#define HEADER_2_H

#include "header-1.h"

int myFunction2(MyStruct *value);

#endif
```

main.c

```c
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

- Choose a “label” (actually a macro) that is unique to each header file.
- Use #ifndef <label> paired with #endif, typically at end of file.
  - If the label has not been defined (header file has not been seen), #define it, and everything that follows will be included.
  - If the label has been defined, everything else before #endif will be skipped!

https://riptutorial.com/c/example/7662/header-include-guards

What gets included

- Here is what the preprocessor actually puts together in this case.
- Note that the second definition of MyStruct will be skipped.

```
#ifndef HEADER_1_H
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

#ifndef HEADER_2_H
#define HEADER_2_H

#ifndef HEADER_1_H // Safe, since HEADER_1_H was #define'd before.
#define HEADER_1_H

typedef struct {
    …
} MyStruct;

int myFunction(MyStruct *value);

#endif

int myFunction2(MyStruct *value);

#endif

int main() {
    // do something
}
```

Included because HEADER_1_H was not defined

Skipped!

main.c

```
#include "header-1.h"
#include "header-2.h"

int main() {
    // do something
}
```

https://riptutorial.com/c/example/7662/header-include-guards

#include guards

- You should use include guards in your header files!
- It is good practice to do so.
- Even if you think you are not at risk, you never know when you might extend your code in such a way that a file may be included more than once.
- You may also consider using `#pragma once` as an alternative but it may not be supported uniformly, so your code may be less portable.

More conditional compilation

- #ifdef, #else, #endif can be used for conditional compilation.
- Example:

```
#ifdef UNDIRECTED
#   include "undirected.h"
#   define SYMMETRIC 1
#else
#   include "directed.h"
#   define SYMMETRIC 0
#endif
```

Executes if UNDIRECTED has been defined by the preprocessor

Executes if UNDIRECTED has not been defined

- One use: debugging statements (see Workshop 3)
- You may also find #if #elif #endif useful (you can look up)

Predefined macros

Macro	Meaning
__DATE__	Character string representing the date of preprocessing (MMM dd yyyy)
__TIME__	Time of translation (hh:mm:ss)
__FILE__	Character string literal representing the name of the current source code file
__LINE__	Integer constant representing the line number in the current source file

- __func__ is a C predefined identifier that expands to a string representing the name of the function that calls it and can be used with the preprocessor macros above.
- __func__ and __LINE__ may be useful for debugging.