Data Structures in C

Prof. Georg Feil

Preprocessor Directives

Winter 2018

Copyright 2016-2018, Georg Feil, Sheridan College
May not be published or modified without permission.

Acknowledgement

- These lecture slides are partly based on slides by Professor Simon Hood
- Additional sources are cited separately

Reading Assignment (required)

- C for Programmers (supplementary textbook)
  - Chapter 13, section 13.1 to 13.4

The C Preprocessor

- Recall that lines starting with **#** are handled by the C preprocessor, not the C compiler itself
  - The preprocessor is a text processor which runs before the compiler
  - The preprocessor's output is fed to the compiler
- The preprocessor supports commands or "directives" to
  - Include other files in the file being compiled
  - Define symbolic constants and macros
  - Perform conditional compilation of code, including conditional execution of other preprocessor directives
  - Control compiler behaviour or options (#pragma and #error)

#include directive

- We've learned this one already…
  - The specified file is inserted at that point
- There are two types of include

```
#include <string.h>
```

  - This means look in system header file directories for the file

```
#include "myFile.h"
```

  - This also looks in the same directory as the file being compiled
- Note you can use (relative) path names if you need

```
#include <sys/time.h>
#include "../h/myFile.h"
```

#define directive

- This directive defines a new symbol to be used as a constant value or macro

```
#define symbol replacement
```

- After this definition any time the preprocessor sees 'symbol' anywhere it replaces it with 'replacement'
  - Remember this happens before the program is compiled

```
#define PI 3.1415926536
```

No semicolon

- The example above shows how to use #define to create a constant for π in your program
  - The code `tau = 2*PI` will become `tau = 2*3.1415926536`

#define directive – dangers

- You should be very careful when using #define
  - It can mangle your code in horrible ways
- For example suppose you did this

```
#define if maybe
```

- After this all 'if' statements are broken… syntax errors!
- To help avoid problems you should never #define symbols which might be keywords or variable names
  - C programmers almost always use ALL_UPPER_CASE symbols with #define to avoid problems

Should I use a constant variable or macro?

- In C there are two options when defining a constant

```
const double Pi 3.1415926536;
#define PI 3.1415926536
```

- You may see #define used for constants in many older C programs
- The const method is more modern
- I suggest you use const for constant values accessed within a single source file, when possible
  - const variables shared by multiple source files can run into linker issues unless you do it right, #define is an easy way out
  - Sometimes array declarations aren't allowed with const variables

Preprocessor macros with parameters

- #define symbols can also accept parameters
- In this case it's called a macro
- Macro parameters work by simple text substitution, for example

```
#define CIRCLE_AREA(x) ( PI * (x) * (x) )
```

- This macro "generates" the code needed to calculate the area of 'x'
  - Putting in extra parentheses is recommended or bugs can arise
- Using #define with parameters can lead to many hard-to-find bugs, and should be avoided whenever possible
  - We won't use #define with parameters in this course

Undefining macros

- You can use #undef to “cancel” the definition of a symbol from that point on
- You should use #undef only if really needed
- One possible use is if a header file you include declares a constant/macro that conflicts with your code
  - You can remove the macro, or change the value of a constant to be compatible with your code

Exercise 1

- Recall that in “C Structures” Exercise 6 we created an array of structures
- Update the program so it uses a const variable for the size of the array
- Now change it to use a parameterless macro (#define) instead