Lecture 13

Some miscellaneous topics

Fundamentals of Computer and Programming

Instructor: Morteza Zakeri, Ph.D. (m-zakeri@live.com)

Spring 2024

Modified Slides from Dr. *Hossein Zeinali* and *Dr. Bahador Bakhshi*

Computer Engineering Department, Amirkabir University of Technology

File postfix

- Most compilers consider the source code file postfix:
  - .c → C source code
  - .cc , .cpp → C++ source code
  - .h for header file → C and C++ codes

A program in multiple file

- We can create our ".h" files
- func.c

```
#include <stdio.h>
void f(int x) {
  printf("%d", x);
}
```

- func.h

```
void f(int);
```

- main.c

```
#include "func.h"
int main(void){
  f(20);
}
```

Preprocessor Command

- We can use preprocessor commands to control how our code is compiled
  - Conditional compilation
- Main preprocessor commands
  - #define XYZ → define XYZ as a preprocessor definition (value is not important)
  - #ifdef XYZ → is true if XYZ is defined
  - #ifndef XYZ → is true if XYZ is not defined
  - #if XYZ → is true if XYZ != 0
  - #endif → End of a if block

Preprocessor Command

```
#include <stdio.h>
#define ABC
#define XYZ 1

int main(){
#ifdef ABC
  printf("ABC is defined \n");
#endif
  printf ("I am here\n");
#if XYZ
  printf("XYZ is defined and is not 0\n");
#endif
}
```

Use Preprocess Commands for Debugging

```
#include <stdio.h>
#define DEBUG 1

int f(int x){
#if DEBUG
    printf("We are in file = %s, in function %s, in line %d\n",
   __FILE__, __func__, __LINE__);
#endif
    return x;
}

int main(void){
#if DEBUG
    printf("We are in file = %s, in function %s, in line %d\n",
   __FILE__, __func__, __LINE__);
#endif
    f(10);
    getchar();
    return 0;
}
```

Formatting, Naming, Documenting

- Be consistent with the *formatting* of the source code (e.g., indentation strategy, tabs versus spaces, spacing, brackets/parentheses).
- Avoid a formatting style that runs against common practices.
- Be consistent in the *naming conventions* used for identifiers (e.g., names of objects, functions, namespaces, types) and files.
- Avoid bizarre naming conventions that run against common practices.
- *Comment* your code. If code is well documented, it should be possible to quickly ascertain what the code is doing without any prior knowledge of the code.
- Use *meaningful names* for identifiers (e.g., names of objects, functions, types, etc.). This improves the readability of code.
- Avoid *magic literal constants*. Define a constant object and give it a meaningful name.

```
constexpr double miles_per_kilometer = 0.621371;
```

Error Handling

- If a program requires that certain *constraints on user input* be satisfied in order to work correctly, do not assume that these constraints will be satisfied. Instead, always check them.
- Always handle errors *gracefully*.
- Provide *useful* error messages.
- Always *check return codes*. Even if the operation/function theoretically cannot fail (under the assumption of bug-free code), in practice it may fail due to a bug.
- If an operation is performed that can fail, check the *status of the operation* to ensure that it did not fail (even if you think that it should not fail). For example, check for error conditions on streams.
- If a function can fail, always check its *return value*.

Simplicity

- Do not ***unnecessarily complicate*** code. Use the simplest solution that will meet the needs of the problem at hand.
- Do not impose ***bogus limitations***. If a more general case can be handled without complicating the code and this more general case is likely to be helpful to handle, then handle this case.
- Do not ***unnecessarily optimize*** code. Highly optimized code is often much less readable. Also, highly optimized code is often more difficult to write correctly (i.e., without bugs). Do not write grossly inefficient code that is obviously going to cause performance problems, but do not optimize things beyond avoiding gross inefficiencies that you know will cause performance problems.

Code Duplication

- **Avoid *duplication* of code. If similar code is needed is more than place, put the code in a function. Also, utilize templates to avoid code duplication.**
- **The avoidance of code duplication has many advantages.**

1. It simplifies code understanding. (Understand once, instead of n times.)
2. It simplifies testing. (Test once, instead of n times.)
3. It simplifies debugging. (Fix bugs in one place, instead of n places.)
4. It simplifies code maintenance. (Change code in one place, instead of n places.)

- **Make good use of the available *libraries*. Do not reinvent the wheel. If a library provides code with the needed functionality, use the code in the library.**

The last words …

- Use as many information resources as you can to learn as much as you can about C.
- Read books, articles, and other documents.
- Watch videos.
- Attend lectures and seminars.
- Participate in programming competitions.
- But most importantly:

  Write code!

  Write lots and lots and lots of code!
- The only way to truly learn a programming language well is to use it heavily (i.e., write lots of code using the language).

Reference

- Reading Assignment: Chapters 13 and 14 of “C How to Program”