TEXAS A&M UNIVERSITY
Department of Computer Science & Engineering

CSCE 120/121
Introduction to Program Design & Concepts

Software Development Process
Comments, Organization, Errors, Debugging

Dr. Tim McGuire

Grateful acknowledgment to Dr. Philip Ritchey and Dr. Michael Moore for some of the material on which these slides are based.

Good code is self-documenting.

- Oft-repeated
- Seldom understood

- Good code reduces the need for comments
  - Good code still has comments.
  - Really good code has really good comments.

Comments

- Used to document parts of the program
- Intended for persons reading the source code of the program:
  - Indicate the purpose of the program
  - Describe the use of variables
  - Explain complex sections of code
- Are ignored by the compiler

The are essentially 2 types of comments

- Documentation
  - intended for those who would use your code, but not likely read it
  - document how to use the code (input, output, errors/exceptions, etc.)
- Clarification
  - Intended for those who would maintain and extend the code
  - Often a code smell for overly complex code
    - Try to simplify the code and make it more self-documenting
    - Exceptions
      - The solution is non-intuitive and needs context
      - The solution looks like it can be improved but experience shows that it cannot

Code tells you how, comments tell you why

- Simplify the code until it is as easy to understand as possible
- Explain the remaining complexity with comments
  - Including the rationale for non-obvious design/implementation decisions

```
# reverse the string
string = ''.join([c for c in reversed(string)])
string = string[::-1]
```

The only reason to add comments is to explain something which is not obvious from the code.

Comments should **complement** good coding style, not replace it

The better written your code, the fewer comments you will need

- The only reason to add comments is to explain something which is not obvious from the code.
- Comments should **complement** good coding style, not replace it
  - The better written your code, the fewer comments you will need

- Don't say something obvious, like

```
count = 0;      // store zero in count
```

- Instead, put the instruction into the context of the program

```
count = 0;      // count counts terms, initially 0
```

Error Handling

Types of Errors

- Syntax
- Linker
- Run-time
- Logical

Syntax Errors

- Violation of C++ syntax rules
- Compile-time errors
- Most common:
  - Missing a symbol, e.g. one of: , . ( ) { } [ ] ; + - * /
    - Most common: ; and }
  - Use before declaration
  - Putting a symbol where it doesn't belong
  - 5 = x;

Linker Errors

- Occur after compilation (during linking)
- Executable file cannot be produced
- Most common:
  - Missing header file
  - Incorrect header file
  - Incorrectly named function, e.g. `Main` instead of `main`

Runtime Errors

- Occur during program execution
- Not caught by compiler (usually no line number information)
- Most common:
  - Divide by 0
  - Attempt to access inaccessible memory
    - Null pointer dereference
    - Segmentation Fault
  - Out of memory

Logical Errors

- Occur during execution
- Program runs, but gives incorrect result
- Most common:
  - Off by one
  - Using = instead of == in a conditional
  - Infinite loops
  - Integer division
  - Array access out of bounds
  - Uninitialized variables

What to do when there's an error

- If there is an error message, READ IT
  - Compile-time errors will tell you what the error was, which file contains the error and the number of the line at which the error was discovered.
  - Linker errors will tell you what thing couldn't be found, from which you can often figure out why it couldn't be found.
  - Runtime errors will tell you what went wrong, from which you can narrow down to where in the code the kind of thing that went wrong could have gone wrong.
  - If you don't understand the error message, ask for help understanding the error
- Start debugging

Organization of Code

The Fundamental Theorem of Formatting

Good visual layout shows the logical structure of a program.

Programming Style

- The visual organization of the source code
- Includes the use of spaces, tabs, and blank lines
- Does not affect the syntax of the program
- Affects the readability of the source code

What is the value of x?

```
int x = 0;
for (int i = 0; i < 4; i++)
    x += i;
    x *= 2;
    x--;
```

What is the value of x?

```
int x = 1+5 * 8+1;
```

Objectives of good layout

- Accurately represent the logical structure of the code
- Consistently represent the logical structure of the code
- Improve readability
- Withstand modifications

Whitespace and Parentheses

- Usewhitespacetoenhancereadability.
  - Grouping: group related statements together into “paragraphs”
  - Blank lines: separate unrelated statements from each other
    - Optimal amount of blank lines is between 8% and 16%
  - Indentation: indent statements under the statement to which they are subordinate
    - 20%-30% improvement in code comprehension over no indentation
    - 2 or 4 spaces of indentation (per level) is optimal
- (you should) use more parentheses (than you think (you need)).
  - `cout << 12 + 4 % 3 * 7 / 8. << endl;`

Laying out files and programs

- Put one class in two files
  - Declarations in .h
  - Definitions in .cpp
- Give the file a name related to the class name
  - Typically the same name, e.g. ClassName.cpp and ClassName.h
- Separate methods within a file clearly
  - 2 blank lines between methods
- Order methods alphabetically

Layout of a typical production source file

/* file description comment */
#include files
using-declarations
constant definitions that apply to more than one class
enums that apply to more than one class
macro function definitions
type definitions that apply to more than one class
global variables and functions imported
global variables and functions exported
classes
variables and functions that are private to the file

Layout of a typical source file in 121


```
/* file description comment */
#include files
using-declarations
constant definitions that apply to more than one class
enums that apply to more than one class
macro function definitions
type definitions that apply to more than one class
global variables and functions imported
global variables and functions exported
classes
variables and functions that are private to the file
```

Layout of a typical source file in 121


```
/* file description comment */
#include files
using-declarations
constant definitions that apply to more than one class
classes
variables and functions that are private to the file
```

Layout of a typical source file in 121


```
/* file description comment */
#include files
using-declarations
~~constant definitions that apply to more than one class~~
classe~~s~~
variables and functions that are private to the file
```

Layout of a typical source file in 121


```
/* file description comment */
#include files
using-declarations
class
variables and functions that are private to the file
```

Laying out functions

- Use blank lines to separate parts of a method
- Use standard indentation for function parameters

```
void quadratic(
    double a,
    double b,
    double c)
{
    // input validation

    // processing
}
```

Laying out comments

- Indent a comment with its corresponding code
- Set off each comment with at least one blank line

```
// comment
code;

// comment
while (condition) {
    // comment
    code;
}
```

Laying out individual statements

- Limit line length to about 80
  - OK to exceed occasionally with good reason
- Use spaces to make logical expressions, array references, and method arguments readable
  - `if ((a && b) || (c < d))`
  - `array[ index ]`
  - `foo(arg1, arg2, arg3)`
- Indent continuation lines the standard amount
  -
```
int variable = thisVariable
        + thatVariable
        - anotherVariable;
```

Laying out individual statements

- One statement per line
- Avoid using multiple operations per line
- Use only one declaration per line
- Declare variables close to where they are used
- Order declarations sensibly
  - e.g. by type

Laying out control structures

- The close **}** should be lined up with the beginning of the block it starts

```
if (condition) {
    do stuff
}
```

or

```
if (condition)
{
    do stuff
}
```

- Use blank lines between paragraphs (segments of related code)

```
block1_1;
block1_2;
block1_3;

block2_1;
block2_3;
```

- Format single-statement blocks consistently

Laying out control structures

- For complicated expressions, put separate conditions on separate lines

```
if ((a && b) ||
    (c && d) ||
    (e && f))
{
    …
}
```

Use a linter

- Cpplint (https://github.com/google/styleguide/blob/gh-pages/cpplint/cpplint.py)
  - This is what Mimir uses
  - Suggested usage:

```
./cpplint.py --filter=-legal,-whitespace/tab,-whitespace/end_of_line,-whitespace/ending_newline,-build/include_subdir
```

TEXAS A&M UNIVERSITY
Department of Computer Science & Engineering

Unit Testing

Dr. Tim McGuire

The gist of testing

- Programming introduces bugs
- Testing helps find bugs
- Debugging finds and fixes bugs
- Many types of testing: different levels of abstraction and methodologies
- Testing measures code correctness/quality
  - You cannot test correctness/quality into code
- You usually can't test all possible inputs or even all code paths
  - Aim to get "enough" code coverage. "Enough" is about 85% - 90%.
  - Quality over quantity
- TDD: Write tests first, then write code to pass the tests
  - Tests are formal expressions of the requirements

Unit Testing

- Unit := “smallest testable part of software”
  - Usually few inputs and a single output
- Unit Test: Test a single unit (in isolation)
- Goal: validate that each unit is correct
  - If units are correct, they can be (correctly) combined to solve problems
- Typically automated
  - Many frameworks exist. Mimir uses Google Test for C++ (https://github.com/google/googletest)
- TDD: write the tests before you write the function