Computer Programming Good Programming Style 3rd Lecture

엄현상 (Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

Outline

- Good Programming Style
- Q&A

Guidelines for Writing C/C++ Code

- Point of a Style Guide
  - Greater Uniformity in Appearance of Source Code
- Benefit
  - Enhanced Readability and Hence Maintainability for the Code

File Contents

- Files as Modules to Group Functionality
  - Avoiding Duplicating Functionality in Separate Files
- Header Files
  - To Declare Public Interfaces
- Code Files
  - To Define Implementations
    - If a module calls a function defined externally, it is desirable to include that function's associated .h file in the implementation of the module

Header (Interface) File Contents

- Copyright Statement Comment
- Module Abstraction Comment
- Revision-String Comment; e.g., Id
- Multiple Inclusion **#ifdef** (a.k.a. "include guard")
- Other Preprocessor Directives, **#include** and **#define**
- C/C++ **#ifdef**

Header File Contents Cont'd

- Data Type Definitions (Classes and Structures)
- **typedefs**
- C/C++ **#endif**
- Multiple Inclusion **#endif**

```
#ifdef __cplusplus // predefined (double underscore)
extern 'C"{ // Linkage directive informs the compiler not to encode f/n
#endif
...
#ifdef __cplusplus
}
#endif
```

gcc/g++ Basic Options

- -D
  - Set the Value of a Symbol
- -I (Capital i)
  - Include Files in a Non-Standard Directory

#define INFO_FILE “infofile”

```
martini:~$ g++ -c -DINFO_FILE=\“infofile\” backup1.C
martini:~$ g++ -c -DUSE_ODIR backup2.C
martini:~$ g++ -c -I../include backup3.C
```

indicate where to find the header files

```
#define USE_ODIR
----------------
#ifdef USE_ODIR
...
#else
...
#endif
```

Code File Contents

- Copyright Statement Comment
- Module Abstraction Comment
- Preprocessor Directives, **#include** and **#define**
- Revision-String Variable
  - Implementation-File Revision String Should Be Stored as a Program Variable

Code File Contents Cont'd

```
static const char rcs_id[] = "$Id$";
```

- Other Module-Specific Variable Definitions
- Local Function Interface Prototypes
- Class/Function Definitions

File Format

- Spatial Structure Illustrating the Logical Structure
  - Blank Lines to Help Separate Different Ideas
  - Indentation to Show Logical Relationships
  - Spaces to Separate Functionality
  - Each Block to Do Exact One Thing

File Format Cont'd

- All Function Definitions and Declarations Starting in Column Zero
  - Return Value Type, Function Interface Signature (Name and Argument List), and Function Body Open and End Brackets Put Each on a Separate Line
- Single Space to Separate All Operators from Their Operands
  - Exceptions: ->, ., () and [] Operators

File Format Cont'd

- Four Spaces for Each Level of Indentation
- Lines with No Longer Than 80 Characters
  - Breaking After a Comma
  - Breaking Before an Operator
  - Breaking Lines to Illustrate their Logical Relationships
  - Aligning the Newline with the Beginning of the Expression at the Same Level on the Previous Line

File Format Cont'd

- Pure-Block, Fully Bracketed Style for Blocks of Code
  - Opening Bracket Put at the End of the Line
    - Exception: conditions that are broken across multiple lines

```
new_shape = affine_transform(coords, translation,
                             rotation);

if ( ( (new_shape.x > left_border) &&
       (new_shape.x < right_border) ) &&
     ( (new_shape.y > bottom_border) &&
       (new_shape.y < top_border) ) )
{
    draw(new_shape);
}
```

Unique to C++

- Starting public, protected, private and friend Labels in Column Zero of Class Declarations
- Declaring the Members in a Consistent Order
- Putting Simple Inline Function Definitions on the Same Line as Their Definitions
  - Using a Pure-Block Style with Four-Space Indentation for Complex Inline Functions
- Avoiding Putting Complex Function Implementations into .h Files

Class Declaration Format

```
class Type : public Parent {
private:
    int x_;
    int y_;
public:
    Type();
    Type(int x) : x_(x) { }
    ~Type();

    int get_x() const { return x_; }
    void set_x(const int new_x) { x_ = new_x; }
    ...
    void display() {
        ...
    }
}
```

Choosing Meaningful Names

- Variable Names
  - Lower Case for All Variable Names with an Underscore as a Separator in C/C++
    - E.g., boiling_point
  - Variable Names Using Mixed Case Letters Starting with a Lower Case Letter And Starting Each Subsequent Word with an Upper Case Letter in Java
    - E.g., boilingPoint

Choosing Meaningful Names Cont'd

- Variable Names Cont'd
  - Careful Choice
    - Consistent names
    - Similar names for similar data types
    - No names that are homophones
    - Names that say what the variable represents; i.e., nouns
    - No generic names such as tmp, buf, and reg
    - No intentionally misspelled words such as lo or lite
    - No abbreviations
    - No overly long names

Choosing Meaningful Names Cont'd

- Function Names
  - Lower Case Letters for Public Function Names with an Underscore as a Separator
  - Consistent and Informative Names
    - Strong verb that indicates the purpose for a function that returns no value
    - Name that indicates the meaning of the value returned for a function that returns a value
- Method Names
  - Method Names Using Mixed Case Letters Starting with a Lower Case Letter And Starting Each Subsequent Word with an Upper Case Letter

Choosing Meaningful Names Cont'd

- Classes, Structures, and Type Definitions
  - Capitalizing the First Letter of the Name of Each Type That Is Defined
- Constants
  - Using ALL_UPPER_CASE for Your Named Constants, Separating Words with the Underscore Character

Comments

: Describing *Why* Code Does What It Does

- End-Line Comments
  - Variable Declarations
  - Marking **#if/#endif** Statements
- Short (Single-Line) Comments
- Block Comments
  - Function Descriptions
- Bold Comments
  - Delimiting Major Sections of Code

Illustrations: Comments

```
^L
/*
 * ************************************************
 * Bold comment.
 * ************************************************
 */

/*
 * Block comment.
 */

/* Short (single-line) comment. */
```

```
int i;  /* end-line comment */
```

Syntax and Language Issues

- Each Line to Do Exact One Thing
- No Use of Side-Effects
- Clear Structure
- Trivial Branch
- **while**() { ... } Rather Than **do** { ... } **while** ();
- Short Control Structure
- No Deeply Nested Code
- No Use of Global Variable

Syntax and Language Issues Cont'd

- No Preprocessor Constants (#defines)
  - Declaring Vars of Proper Types as consts
  - Defining enums for Related Sets of Integer Constants
- Function Declarations/Prototypes for All Functions
- Explicit Assumptions about the Condition of Input Data to Routines
- Checking the Return Values of All Library Function Calls
- Informative Error Messages

Formatting

- Formatting Refers to the Indentation, Alignment, And Use of White Space to Lay Out Your Program to Increase Its **Readability** by Others
- **Consistency** Is the Key to Producing Readable Code
  - While Many Can Argue to Merits of 3 Versus 4 Spaces of Indentation, Placement of Curly Braces, Etc.

Real Key Is to Adopt a Formatting Style And Keep to It!