ADVANCED PROGRAMING PRACTICES

Coding conventions

Coding conventions

- **What is it?**
- Coding conventions are a set of prescriptive rules that pertain to how code is to be written, including:
  - **File organization**: how code is distributed between files, and organized within each file.
  - **Indentation**: how particular syntactical elements are to be indented in order to maximize readability.
  - **Comments**: how to consistently and efficiently use comments to help program understandability.
  - **Declarations**: what particular syntax to use to declare variables, data structures, classes, etc. in order to maximize code readability.
  - **Naming**: how to give names to various named entities in a program as to convey meaning embedded into the names.

Coding conventions

- **Who does it?**
  - Coding conventions are applicable to the original programmers and peer reviewers, and eventually the maintainers of a software system.
  - Other workers that are using the code are also likely to be affected, such as testers involved in unit or integration testing.
- **Why do it?**
  - Coding conventions only improve internal qualities of the software and generally do not affect any externally visible quality.
  - Coding conventions aim at maximizing the productivity of the coding process by making code more readable and understandable.
  - Using coding conventions makes it easier to develop further code in a project and eventually aims at increasing the sustainability of the development by decreasing the cost of adding code to an existing code base.

Coding conventions

- **How to do it?**
  - Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual or a group of coders.
  - Can be verified and enforced by a peer review mechanism.
  - Coding conventions are not enforced by compilers, though some IDEs may provide a “pretty printer” feature that will implement some aspects of coding conventions such as indentation.
  - Some code refactoring activities can be used to implement some code changes that are related to coding conventions, such as renaming or breaking larger functions into smaller ones.
  - Another related tool/activity is the use of an automated API documentation tool, which uses specially formatted code comments to provide automatically generated documentation for the code, which also improves software understandability.

Coding conventions

- Compare the two following pieces of code:

```
void calc(double m[], char *g){ double tm = 0.0; for
(int t = 0; t<MAX_TASKS; t++) tm += m[t]; int i =
int(floor(12.0*(tm - MIN) / (MAX - MIN))); strcpy(g, let[i]);}
```

```
void calculateGrade(double marks[], char *grade)
{
    double totalMark = 0.0;
    for (int task = 0; task < MAX_TASKS; task++)
        totalMark += marks[task];
    int gradeIndex = int(floor(12.0 * (totalMark - minMarks) / (maxMarks - minMarks)));
    strcpy(grade, letterGrades[gradeIndex]);
}
```

- The compiler sees these two pieces of code as identical.
- What about a human?

- Code readability is a very important code quality.

Coding conventions

- Three basic rules to increase code readability/understandability:
  - Use a clear and consistent layout.
  - Choose descriptive and mnemonic names for files, constants, types, variables, and functions/methods.
  - Use comments when the meaning of the code by itself is not completely obvious and unambiguous.

Code layout

Code layout

- Code must be indented according to its nesting level.
- The body of a function/method must be indented with respect to its function header; the body of a **`for`**, **`while`**, or **`switch`** statement must be indented with respect to its first line; and similarly for **`if`** statements and other nested structures.
- You can choose the amount of indentation but you should be consistent. A default tab character (eight spaces) is too much: three or four spaces is sufficient.
- Most editors and programming environments allow you to set the width of a tab character appropriately.
- Bad indentation makes a program harder to read and can also be a source of obscure bugs that are hard to locate.

```
while (*p)
  p->processChar();
  p++;
```

Code layout

- One typical point of variation on code layout conventions is how to format statements that use statement blocks.
- Two approaches:
  - Maximize visibility of the different blocks by having curly braces alone on their line of code.
  - Minimize code length by appending the open curly brace to the statement that precedes it.

Code layout

```
Entry *addEntry (Entry * & root, char *name)
   // Add a name to the binary search tree of file descriptors.
{
    if (root == NULL)
    {
       root = new Entry(name);
       if (root == NULL)
          giveUp("No space for new entry", "");
       return root;
    }
    else
    {
       int cmp = strcmp(name, root->getName());
       if (cmp < 0)
          return addEntry(root->left, name);
       else if (cmp > 0)
          return addEntry(root->right, name);
       else
          // No action needed for duplicate entries.
          return root;
    }
}
```

```
Entry *addEntry (Entry * & root, char *name) {
    // Add a name to the binary search tree of file descriptors.
    if (root == NULL) {
        root = new Entry(name);
        if (root == NULL)
            giveUp("No space for new entry", "");
       return root;
    } else {
        int cmp = strcmp(name, root->getName());
        if (cmp < 0)
            return addEntry(root->left, name);
        else if (cmp > 0)
            return addEntry(root->right, name);
        else
            // No action needed for duplicate entries.
            return root;
    }
}
```

Code layout

- For readability purpose, blank lines can be added to separate code components/sections.
- Places where a blank line is often a good idea:
  - between major sections of a long and complicated function.
  - between public, protected, and private sections of a class declaration.
  - between class declarations in a file.
  - between function and method definitions.

Naming conventions

Naming conventions

- Various kinds of names occur within a program:
  - constants;
  - user-defined types, classes;
  - local variables;
  - attributes (data members);
  - functions;
  - methods (member functions).
- It is easier to understand a program if you can guess the “kind” of a name without having to look for its declaration which may be far away or even in a different file.
- There are various conventions for names. You can use:
  - A convention you found and adopted.
  - Your own convention.
  - You may not have an option: some employers require their programmers to follow the company’s style.

Naming conventions

- Generally accepted naming conventions:
  - The length of a name should depend on its scope.
    - Names that are used pervasively in a program, such as global constants, must have long descriptive names.
    - A name that has a small scope, such as the index variable of a one-line for statement, can be short: one letter is often sufficient.
  - Constants are named with all upper case letters and may include underscores.
  - User-defined type names or class names start with a capital letter.
  - Avoid very long names, as they tend to create more multiple-line statements, which are harder to read and understand.
  - Names that contain multiple words are either separated by a delimiter, such as underscore, or by using an upper case letter at the beginning of each new word (CamelCaseNaming).

Example of coding conventions

- Brown University has a set of coding standards used for introductory software engineering courses. Here are a few:
  - **File names use lower case characters only.**
    - UNIX systems distinguish cases in file names: `mailbox.h` and `MailBox.h` are different files. One way to avoid mistakes is to lower case letters only in file names. Windows does not distinguish letter case in file names. This can cause problems when you move source code from one system to another. If you use lower case letters consistently, you should not have too many problems moving code between systems. Note, however, that some Windows programs generate default extensions that have upper case letters!
  - **Types and classes start with the project name.**
    - An abbreviated project name is allowed. For example, if you are working on a project called `MagicMysteryTour`, you could abbreviate it to `MMT` and use this string as a prefix to all type and class names: `MMTInteger`, `MMTUserInterface`, and so on. This may not be necessary for isolated projects. The components of a project are usually contained within a single directory, or tree of directories, and this is sufficient indication of ownership. The situation is different for a library, because it must be possible to import library components without name collisions.

Example of coding conventions

- **Method names start with a lower case letter and use upper case letters to separate words.**
  - Examples: `getScore()`, `isLunchTime()`. Some use this notation for both methods and attributes. In the code, you can usually distinguish methods and attributes because method names are followed by parentheses.
  - This is commonly called “CamelCase”.
- **Attribute names start with a lower case letter and use underscores to separate words.**
  - Examples: `start_time`, `current_task`.
- **Constants use upper case letters with underscores between words.**
  - Examples: `MAXIMUM_TEMPERATURE`, `MAIN_WINDOW_WIDTH`.
- **Global names are prefixed with the project name.**
  - Example: `MMTstandardDeviation`. This may avoid name clashes when the code is combined/reused elsewhere which may have the same global variable names.
- **Function/method’s local variables are written entirely in lower case without underscore.**
  - Examples: `index`, `nextitem`.

Example of coding conventions

- In *Large-Scale C++ Software Design*, John Lakos suggests prefixing all attributes with **`d_`**.
- This has several advantages; one of them is that it becomes easy to write constructors without having to invent silly variations.
- Another similar naming convention is to prefix all constructors' parameter names by **`new_`**.

```
Clock::Clock(int new_hours, int new_minutes, int new_seconds)
{
    d_hours = new_hours;
    d_minutes = new_minutes;
    d_seconds = new_seconds;
}
```

Example of naming conventions

- The **Hungarian notation** was introduced at Microsoft during the development of OS/2.
  - It is called “Hungarian” because its inventor, Charles Simonyi, is Hungarian.
  - Also, identifiers that use this convention are hard to pronounce, like Hungarian words (if you are not Hungarian, that is).
  - If you do any programming in the Windows environment using C++, you will find it almost essential to learn Hungarian notation.
  - Hungarian variable names start with a small number of lower case letters that identify the type of the variable.
  - These letters are followed by a descriptive name that uses an upper case letter at the beginning of each word.
  - For example, a Windows programmer knows that the variable `lpszMessage` contains a long pointer to a string terminated with a zero byte.
  - The name suggests that the string contains a message of some kind.
  - Makes C++ programs more understandable by including the variables’ typing as part of their name, typing being an important problem in C++ programming.
  - Good example of programming language-specific naming convention.
  - The following table shows some commonly used Hungarian prefixes.

Example of naming conventions

c	character
by	unsigned char or byte
n	short integer (usually 16 bits)
i	integer (usually 32 bits)
x, y	integer coordinate
cx, cy	integer used as length (“count”) in X or Y direction
b	boolean
f	flag (equivalent to boolean)
w	word (unsigned short integer)
l	long integer
dw	double word (unsigned long integer)
fn	function
s	string
sz	zero-terminated string
h	handle (for Windows programming)
p	pointer

Commenting conventions

Commenting conventions

- Comments should be used to improve code understandability.
- Comments are an important part of a program but you should not overuse them.
- Overuse of comments may “drown” the code in overabundant comments.
- The following rule will help you to avoid over-commenting:
  - **Comments should not provide information that can be easily inferred from the code.**

Commenting conventions

- There are two ways of applying this rule:
  - To eliminate pointless comments

```
counter++; // Increment counter.

// Loop through all values of index.
for (index = 0; index < MAXINDEX; index++)
{
    //loop code
}
```

  - To improve existing code.

```
int np;     // Number of pixels counted.
int flag;   // 1 if there is more input, otherwise 0.
int state;  // 0 = closed, 1 = ajar, 2 = open.
double xcm; // X-coordinate of centre of mass.
double ycm; // Y-coordinate of centre of mass.
```

```
int pixelCount;
bool moreInput;
enum { CLOSED, AJAR, OPEN } doorStatus;
Point massCentre;
```

    - If code needs to be explained, try to change the code so that it does not require explanations rather than include a comment.

Commenting conventions

- There should usually be a comment of some kind at the following places:
  - At the beginning of each file there should be a comment explaining the purpose of this file in the project. More important in C++, where a file can contain many classes.
  - Each class declaration should be preceded by a comment explaining what the class is for.
  - Each method or function should have comments explaining what it does and how it works, as well as what is the purpose of its parameters.
  - All variable declarations, most importantly class data members, should be appended with a comment describing its role, unless its name makes it obvious.
  - All the preceding can be done in a structured manner using documentation tools such as Javadoc/Doxygen.

Commenting conventions

- In cases where an elaborated algorithm is used in a long function, inline comments should be used to highlight and explain all the important steps of the algorithm.

```
void collide (Ball *a, Ball *b, double time)
{
    // Process a collision between two balls.
    // Local time increment suitable for ball impacts.
    double DT = PI / (STEPS * OM_BALL);
    // Move balls to their positions at time of impact.
    a->pos += a->vel * a->impactTime;
    b->pos += b->vel * a->impactTime;
    // Loop while balls are in contact.
    int steps = 0;
    while (true)
    {
        // Compute separation between balls and force separating them.
        Vector sep = a->pos - b->pos;
        double force = (DIA - sep.norm()) * BALL_FORCE;
        Vector separationForce;
        if (force > 0.0)
        {
            Vector normalForce = sep.normalize() * force;
            // Find relative velocity at impact point and deduce tangential force.
            Vector aVel = a->vel - a->spinVel * (sep * 0.5);
            Vector bVel = b->vel + b->spinVel * (sep * 0.5);
            Vector rVel = aVel - bVel;
            Vector tangentForce = rVel.normalize() * (BALL_BALL_FRICTION * force);
            separationForce = normalForce + tangentForce;
        }
        // Find forces due to table.
        Vector aTableForce = a->ballTableForce();
        Vector bTableForce = b->ballTableForce();
        if ( separationForce.iszero() &&
             aTableForce.iszero() &&
             bTableForce.iszero() &&
             steps > 0)
           // No forces: collision has ended.
           break;
        // Effect of forces on ball a.
        a->acc = (separationForce + aTableForce) / BALL_MASS;
        a->vel += a->acc * DT;
        a->pos += a->vel * DT;
        a->spin_acc = ((sep * 0.5) * separationForce + bottom * aTableForce) / MOM_INT;
        a->spinVel += a->spin_acc * DT; a->updateSpin(DT);
        // Effect of forces on ball b.
        b->acc = (- separationForce + bTableForce) / BALL_MASS;
        b->vel += b->acc * DT; b->pos += b->vel * DT;
        b->spin_acc = ((sep * 0.5) * separationForce + bottom * bTableForce) / MOM_INT;
        b->spinVel += b->spin_acc * DT;
        b->updateSpin(DT);
        steps++;
    }
    // Update motion parameters for both balls.
    a->checkMotion(time);
    b->checkMotion(time);
}
```

Summary

- Coding conventions include:
  - Code layout conventions that aim at increasing code readability.
  - Naming conventions that aim at increasing code understandability.
  - Commenting conventions that aim at increasing code understandability.
  - Other coding conventions that aim at:
    - Avoiding certain pit-traps related to either a certain language or operating system.
    - Providing constraints in the use of an overly-permissive language.
- Overall, coding conventions are used to:
  - Increase coding productivity.
  - Decrease the time required to browse through, read, and understand code.
- Requires discipline and rigor.

In the project

- You are required to use the following coding conventions:
  - variable names
    - class names in CamelCase that starts with a capital letter
    - data members start with `d_`
    - parameters start with `p_`
    - local variables start with `l_`
    - consistent layout throughout code (use an IDE auto-formatter)
  - comments
    - javadoc comments for every class and method
    - long methods are documented with comments for procedural steps
    - no commented-out code
  - project structure
    - one folder for every module in the high-level design
    - tests are is a separate folder that has the exact same structure as the code folder
    - 1-1 relationship between tested classes and test classes.

References

- Robert L. Glass: *Facts and Fallacies of Software Engineering*; Addison Wesley, 2003. ISBN-13: 978-0321117427.
- Oracle Corporation. *Code Conventions for the Java Programming Language*.
- Google Inc. *Google Java Style.*
- Peter Grogono. *Course notes for COMP6441: Advanced Programming Practices.* Concordia University, 2007.