Stanford | ENGINEERING
Computer Science

CS 106B: Programming Abstractions (C++), Autumn 2015

NOTE: This web site is **out of date**. This is the course web site from a past quarter, **Autumn 2015**. If you are a current student taking the course, this is not your class web site, and you should visit the current class web site instead at http://cs106b.stanford.edu/. If you are already at cs106b.stanford.edu, the web page may not be updated yet for the new quarter. Please be advised that courses change with each new quarter and instructor. Any information on this out-of-date page may not apply to you this quarter.

CS106B Style Guide

These are some of the general style qualities that we expect your programs to have in order to receive full credit. This is not an exhaustive list; please also refer to each assignment spec for other style practices to follow. Certainly it is possible to write good code that violates these guidelines, and you may feel free to contact us if you are unclear about or disagree with some of them. But we do expect you to follow these rules (unless there is an error in this document). In most professional work environments you are expected to follow that company's style standards. Learning to carefully obey a style guide, and writing code with a group of other developers where the style is consistent among them, are valuable job skills.

This document is a work in progress.
Any guidelines written here are in addition to what is mentioned in the given assignment's spec, so you are responsible for reading that spec and following its instructions. If there is ever a conflict between this style guide and an assignment spec, follow the assignment spec.

Last updated Wed 2014/10/29

Whitespace and Indentation

Indenting: Increase your indentation by one increment on each brace {, and decrease it once on each closing brace }.

Place a line break after every { .

Do not place more than one statement on the same line.

```
// bad
int x = 3, y = 7;  double z = 4
if (a == b) { foo(); }
```

```
// good
int x = 3;
int y = 7;
double z = 4.25;

x++;
if (a == b) {
    foo();
}
```

Long lines: When any line is longer than 100 characters, break it into two lines by pressing Enter after an operator and resuming on the next line. Indent the trailing second part of the line by two increments (e.g. two tabs). For example:

```
int result = reallyLongFunctionOne() + reallyLongFunctionTwo()
        reallyLongFunctionThree() + reallyLongFunctionFour();

int result2 = reallyLongFunction(parameterOne, parameterTwo, p
        parameterFour, parameterFive, parameterSix);
```

Expressions: Place a space between operators and their operands.

```
int x = (a + b) * c / d + foo();
```

Blank lines: Place a blank line between functions and between groups of statements.

```
void foo() {
    ...
}
                              // this blank line here
void bar() {
    ...
}
```

Naming and Variables

Names: Give variables descriptive names, such as `firstName` or `homeworkScore`. Avoid one-letter names like `x` or `c`, except for loop counter variables such as `i`.

Capitalization: Name variables and functions with camel-casing `likeThis`, name classes with Pascal casing `LikeThis`, and name constants in uppercase `LIKE_THIS`.

Scope: Declare variables in the narrowest possible scope. For example, if a variable is used only inside a specific `if` statement, declare it inside that `if` statement rather than at the top of the function or at the top of the file.

Types: Choose appropriate data types for your variables. If a given variable can store only integers, give it type `int` rather than `double`.

Favor C++ strings over C strings: C++ confusingly features two kinds of strings: the `string` class from C++, and the older `char*` (array of characters) from C. As much as possible, you should use the C++ `string` type over the older C string type.

```
// bad: C-style string
char* str = "Hello there";
```

```
// good: C++-style string
string str = "Hello there";
```

Constants: If a particular constant value is used frequently in your code, declare it as a `const` constant, and always refer to the constant in the rest of your code rather than referring to the corresponding value.

```
const int VOTING_AGE = 18;
```

Avoid global variables: Never declare a modifiable global variable. The only global named values in your code should be `const` constants. Instead of making a value global, pass it as a parameter and/or return it as needed.

```
// bad
int count;  // global variable

void func1() {
    count = 42;
}

void func2() {
    count++;
}

int main() {
    func1();
    func2();
}
```

```
// better
int func1() {
    return 42;
}

void func2(int& count) {
    count++;
}

int main() {
    int count = func1();
    func2(count);
}
```

Basic C++ Statements

favor C++ idioms over C idioms: Since C++ is based on C, there is often a "C++ way" to do a given task and also a "C way" to do a given task. For example, when printing output to the system console, the "C++ way" is to use the global output stream `cout`, while the "C way" is to use global functions like `printf`. You should always favor the "C++ way" when possible.

```
// bad
printf("Hello, world!\n");
```

```
// good
cout << "Hello, world!" << endl
```

for vs while: Use a `for` loop when the number of repetitions is known (definite); use a `while` loop when the number of repetitions is unknown (indefinite).

```
// repeat exactly 'size' times
for (int i = 0; i < size; i++) {
    ...
}

// repeat until there are no more lines
string str;
while (input >> str) {
    ...
}
```

break and continue: In general, you should avoid using the `break` or `continue` statements in loops unless absolutely necessary.

`exit()`: C++ contains an `exit` function that immediately exits your entire program. You should never call this function in our assignments. Your program should always

exit naturally by reaching the end of your `main` function and returning.

always include {} on control statements: When using control statements like `if/else`, `for`, `while`, etc., always include {} and proper line breaks, even if the body of the control statement is only a single line.

```
// bad
if (size == 0) return;
else
    for (int i = 0; i < 10; i++
```

```
// good
if (size == 0) {
    return;
} else {
    for (int i = 0; i < 10; i++
        cout << "ok" << endl;
    }
}
```

if/else patterns: When using `if/else` statements, properly choose between various `if` and `else` patterns depending on whether the conditions are related to each other. Avoid redundant or unnecessary `if` tests.

```
// bad
if (grade >= 90) {
    cout << "You got an A!";
}
if (grade >= 80 && grade < 90)
    cout << "You got a B!";
}
if (grade >= 70 && grade < 80)
    cout << "You got a C!";
}
...
```

```
// good
if (grade >= 90) {
    cout << "You got an A!";
} else if (grade >= 80) {
    cout << "You got a B!";
} else if (grade >= 70) {
    cout << "You got a C!";
}
...
```

Boolean zen 1: If you have an `if/else` statement that returns a `bool` value based on a test, just directly return the test's result instead.

```
// bad
if (score1 == score2) {
    return true;
} else {
    return false;
}
```

```
// good
return score1 == score2;
```

Boolean zen 2: Don't ever test whether a `bool` value is == or `!=` to `true` or `false`.

```
// bad
if (x == true) {
    ...
} else if (x != true) {
```

```
// good
if (x) {
    ...
} else {
```

```
    ...
}
```

```
    ...
}
```

Redundancy

Minimize redundant code: If you repeat the same code two or more times, find a way to remove the redundant code so that it appears only once. For example, place it into a helper function that is called from both places. If the repeated code is nearly but not entirely the same, try making your helper function accept a parameter to represent the differing part.

```
// bad
foo();
x = 10;
y++;
...

foo();
x = 15;
y++;
```

```
// good
helper(10);
helper(15);
...

void helper(int newX) {
    foo();
    x = newX;
    y++;
}
```

if/else factoring: Move common code out of `if/else` statements so that it is not repeated.

```
// bad
if (x < y) {
    foo();
    x++;
    cout << "hi";
} else {
    foo();
    y++;
    cout << "hi";
}
```

```
// good
foo();
if (x < y) {
    x++;
} else {
    y++;
}
cout << "hi";
```

Function structure: If you have a single function that is very long, break it apart into smaller sub-functions. The definition of "very long" is vague, but let's say a function longer than 40-50 lines is pushing it. If you try to describe the function's purpose and find yourself using the word "and" a lot, that probably means the function does too many things and should be split into sub-functions.

Efficiency

Save expensive call results in a variable: If you are calling an expensive function and

using its result multiple times, save that result in a variable rather than having to call the function multiple times.

```
// bad
if (reallySlowSearchForIndex("a
    remove(reallySlowSearchFor
}
```

```
// good
int index = reallySlowSearchFor
if (index >= 0) {
    remove(index);
}
```

Comments

Class header: Place a descriptive comment heading on the top of every file describing that file's purpose. Assume that the reader of your comments is an intelligent programmer but not someone who has seen this assignment before. Your comment header should include at least your name, course/section, and a brief description of the assignment. If the assignment asks you to submit multiple files, each file's comment header should describe that file/class and its main purpose in the program.

Citing sources: If you look at *any* resources that help you create your program (a book, lecture slides, section example, web page, another person's advice, etc.), you should list all of them in your comments at the start of the file. When in doubt about whether to cite a source, be liberal and cite it. It is important to cite all relevant sources

Function/constructor headers: Place a comment heading on each constructor and function of your file. The heading should describe the function's behavior.

Parameters/return: If your function accepts parameters, briefly describe their purpose and meaning. If your function returns a value, briefly describe what it returns.

Preconditions/assumptions: If your function makes any assumptions, such as assuming that parameters will have certain values, mention this in your comments.

Exceptions: If your function intentionally throws any exceptions for various expected error cases, mention this in your comments. Be specific about what kind of exception you are throwing and under what conditions it is thrown. (e.g. "Throws an IllegalArgumentException if the student ID passed is negative.")

Inline comments: Inside the interiors of your various functions, if you have sections of code that are lengthy or complex or non-trivial, place a small amount of inline

comments near these lines of complex code describing what they are doing.

Implementation details: Comment headers at the top of a function, class, or file should describe the function's behavior, but not great detail about how it is implemented. Do not mention language-specific details like the fact that the function uses a `if/else` statement, that the function declares an array, that the function loops over a list and counts various elements, etc.

Wording: Your comment headers should be written in **complete sentences**, and should be written in **your own words**, not copied from other sources (such as copied verbatim from the homework spec document).

TODOs: You should remove any `// TODO:` comments from a program before turning it in.

Commented-out code: It is considered bad style to turn in a program with chunks of code "commented out". It's fine to comment out code as you are working on a program, but if the program is done and such code is not needed, just remove it.

Here is a decent overall example of a good comment header on a function. Not every comment header needs to be this long, but since this function takes a parameter and returns something, it needs to mention several things.

```
class Person {
    public:
        bool engageTo(Person& other);
    ...
}

/*
 * Sets this person to be engaged to the given other person.
 * If either this person or other were previously engaged, their previous
 * engagement is called off and the previous partner is set to be single.
 * Returns true if this person was previously engaged before the call.
 * Assumes that other != null and other is of the opposite gender.
 */
bool Person::engageTo(Person& other) {
    ...
}
```

Functions and Procedural Design

Designing a good function: A well-designed function exhibits properties such as the following:

Fully performs a single coherent task.
Does not do too large a share of the work.
Is not unnecessarily connected to other functions.
Stores data at the narrowest scope possible.
Helps indicate and subdivide the structure of the overall program.
Helps remove redundancy that would otherwise be present in the overall program.

Value vs. reference parameters: Use reference parameters to send information 'out' from a function, or when the function may want to change the value of the parameter passed in, or when the function needs to return multiple values. Don't use reference parameters when it is not necessary or beneficial. Notice that a, b, and c are not reference parameters in the following function because they don't need to be.

```
/*
 * Solves a quadratic equation ax^2 + bx + c = 0,
 * storing the results in output parameters root1 and root2.
 * Assumes that the given equation has two real roots.
 */
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double d = sqrt(b * b - 4 * a * c);
    root1 = (-b + d) / (2 * a);
    root2 = (-b - d) / (2 * a);
}
```

Reference 'out' parameter vs. return: When a single value needs to be sent back from a function and it could be provided by a reference 'out' parameter or a return value, favor using a return value.

```
// bad
void max(int a, int b, int& re
    if (a > b) {
        result = a;
    } else {
        result = b;
    }
}
```

```
// good
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Pass objects by reference: When sending an object as a parameter to a function, you should usually pass it by reference because if it is passed by value, the entire object must be copied. Copying objects is expensive.

```
// bad
void process(BankAccount accou
    ...
}

void computeSomething(Vector<Fo
```

```
// good
void process(BankAccount& accou
    ...
}

void computeSomething(Vector<Fo
```

```
    ...
}
```

```
    ...
}
```

Reference vs. pointer: If you have learned about C/C++ pointers from your previous programming experience, favor passing references rather than pointers as much as possible in most cases. (CS 106B covers pointers later in the course.) One reason for this is because a reference, unlike a pointer, cannot be `NULL`.

```
// bad
// accepts a pointer to an acc
void process(BankAccount* accou
    ...
}
```

```
// good
// accepts a reference to an ac
void process(BankAccount& accou
    ...
}
```

Functions that create/build collections: If you have a function whose task is to fill in the contents of a collection, such as a function to read an input file and store the lines into a `Vector`, the preferred C++ way to do this is to have the client pass in a reference to the collection for your function to fill, rather than having your function create and return the collection. This is because returning an entire collection makes a full copy of the state of that collection, which is inefficient.

```
// bad
Vector<string> readFile(string
    Vector<string> v;
    // put some data in v
    ...
    return v;
}
```

```
// good
void readFile(string filename, V
    // put some data in v
    ...
}
```

const reference parameters: If you are passing an object to a function and your code will not modify the state of that object, pass it as a `const` reference.

```
// bad
// accepts a pointer to an acc
void display(BankAccount accour
    ...
}
```

```
// good
// accepts a const reference to
void display(const BankAccount&
    ...
}
```

Avoid "chaining" calls, where many functions call each other in a chain without ever returning to `main`. Make sure that main is a concise summary of your overall program. Here is a rough diagram of call flow with (left) and without (right) chaining:

```
// bad
main
|
+-- function1
    |
    +-- function2
```

```
// good
main
|
+-- function1
|
+-- function2
```

```
        |
        +-- function3
            |
            +-- function4
            |
            +-- function5
                |
                +-- function6
```

```
|   |
|   +-- function3
|       |
|       +-- function4
|
+-- function5
|   |
|   +-- function6
```

Class Design

Encapsulation: Properly encapsulate your objects by making any data fields in your class `private`.

```
class Student {
private:
    int homeworkScore;
        ...
```

.h vs. .cpp: Always place the declaration of a class and its members into its own file, *`ClassName`*`.h`. Place the implementation bodies of those members into their own file, *`ClassName`*`.cpp`. Always wrap the `.h` file's class declaration in an `#ifndef/define/endif` preprocessor block to avoid multiple declarations of the same class.

```
// Point.h
#ifndef _point_h
#define _point_h
class Point {
public:
    Point(int x, int y);
    int getX() const;
    int getY() const;
    void translate(int dx, int dy);

private:
    int m_x;
    int m_y;
};
#endif
```

```
// Point.cpp
#include "Point.h"

Point::Point(int x, int y) {
    m_x = x;
    m_y = y;
}

void Point::translate(int dx, int dy) {
```

```
    m_x += dx;
    m_y += dy;
}
...
```

class vs. struct: Always favor using a `class` unless you are creating a very small and simple data type that just needs a few public member variables and perhaps a constructor to initialize them. Examples of such small `struct` types might be `Point` or `LinkedListNode`.

Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used within a single call to one function.

Helper functions: If you add a member function to a class that is not part of the homework spec, make it `private` so that other external code cannot call it.

```
class Student {
    ...
private:
    double computeTuitionHelper();
```

`const` members: If a given member function does not modify the state of the object upon which it is called, declare it `const`.

```
class Student {
public:
    int getID() const;
    double getGPA(int year) const;
    void payTuition(Course& course);
    string toString() const;
    ...
```