CS180 C++ Language Coding Style Guide

星期一, 八月 19, 2019 11:37 上午

Introduction

A style guide is a set of mandatory requirements for layout and formatting. Uniform style makes it easier for you to read your instructor's code, and for him to read yours. A style guide makes you a more productive programmer because it *reduces gratuitous choice.* If you don't have to make choices about trivial matters, you can spend your energy on the solution of real problems. [From Horstmann.]

Source Files

Each C++ program is a collection of one or more source files.
Organize the material in each file as follows:

- a comment explaining the purpose of this file, along with your full preferred name
- includes, if any
- a namespace statement, if used
- typedefs, defines, and global constants, if any
- function prototypes (with documentation)
- the main function definition, if present
- other function definitions, if present

Lexical Issues

Naming Conventions

- All variable and function names start with a lowercase letter.
  Research shows that lower case with underscores (first_player) cause fewer errors than CamelCase (firstPlayer).
- All constants are in uppercase with UNDER_SCORE characters; for example, CLOCK_RADIUS.
  Names must be long enough to be descriptive.
  Use first_player instead of fp. Local variables that are loop control variables can be single letters (eg., i). Otherwise, do not use names such as ctr, c, cntr, cnt, c2 for variables. Surely these variables all have specific purposes and can be named to explain their purpose (for example, counter, score_counter, and student_counter).

Indentation and White Space

Use two spaces for indenting, with no tab characters.

Use two spaces for indenting, with no tab characters.
Use blank lines freely to separate sections of code that are logically distinct.
Use a blank space around every binary operator (except the selection operators -> and . which we'll study later).
x->value=(-b-sqrt(b*b-4*a*c))/(2*a); // bad
x->value = (-b - sqrt(b * b - 4 * a * c)) / (2 * a); // good
Do not leave a blank space between a unary operator and its argument.
Leave a blank space after, but not before, each comma or semicolon, before an open parenthesis, and after a close parenthesis in an expression. Do not leave a space after a (, {, or [or before a), }, or] in an expression.
Leave a space before but not after the opening paren of if, while, and for statements. Do not leave a space before the opening paren of function calls.

```
if (x == 0)
{
  y = 0;
}
foo(a, (b + 2) / 3);
```

The maximum length for any line is 78 characters. If you must break a statement, add an indentation level for the continuation, and end the previous line with an operator or separator if possible

```
a[n] = ..................................................... +
       .................;
```

Braces

Opening and closing braces must line up vertically in line with the control construct whose block they form. On any line there can be at most one brace.

```
while (i < n) { cout << a.at(i); i++; } // bad
while (i < n) // good
{
  cout << a.at(i);
  i++;
}
```

If-Else and Do-While

The i, es, and braces of if-else structures must be vertically aligned, each by itself on a line

```
if (x == 0) { // bad
  foo( );
```

```
} else {
  bar( );
}
if (x == 0) // good
{
  foo( );
}
else
{
  bar( );
}
```

The do-while structure should be formatted this way:

```
do
{
  foo( );
} while (x == 0);
```

Unstable Layout

Some programmers take pride in lining up certain columns in their code:

```
first_record = some_value;
last_record  = some_other_value;
cutoff       = a_third_value;
```

This is superficially neat, but the layout is not stable under change. A new variable name that is longer than the preallotted number of columns requires that you move *all* entries around:

```
first_record         = some_value;
last_record          = some_other_value;;
cutoff               = a_third_value;
marginal_fudge_factor = one_more_value;
```

This is a trap that leads you to decide to use a short variable name like mff. Instead use a simple layout that is easy to maintain as your program changes.

```
first_record = some_value;
last_record = some_other_value;
cutoff = a_third_value;
marginal_fudge_factor = one_more_value;
```

Every function must have a comment explaining the purpose of the function and should be in Javadoc format. Start with /**, document every parameter and the return value, and finish with */.

Parameter names must be explicit and obvious:

```
void remove(unsigned d, double s) // bad
```

```
void remove(unsigned d, double s); // bad
void remove(unsigned department, double severance_pay)  // good
```

Variables and Constants

Do not define all variables at the beginning of a block. Instead, define each variable just before it is used.

```
{
  double xold; // Don't
  double xnew;
  bool done;
  . . .
}
{
  . . .
  double xold = foo(input);
  bool done = false;
  while (!done)
  {
    double xnew = (xold + a / xold) / 2;
    . . .
  }
  . . .
}
```

Do not define two or more variables in one definition. Instead, use two separate definitions.

```
unsigned dimes = 0, nickels = 0; // Don't
unsigned dimes = 0; // OK
unsigned nickels = 0;
```

Do not initialize variables unnecessarily. If the first use of a variable sets its value, it does not need to be initialized.

```
unsigned foo = 0; // useless, immediately overwritten
while (!done)
{
  cin >> foo;
  ...
```

Do not use magic numbers. A *magic number* is a numeric constant embedded in code, without a constant definition. Except in rare cases (e.g., using 3 to determine whether a number is divisible by 3), every number except –1, 0, 1, and 2 is considered magic. Sometimes even these are magic, if used as codes standing for some value. Use named constants instead.

```
if (p.get_x( ) < 300) // Don't
```

```
const double WINDOW_WIDTH = 300;
. . .
if (p.get_x( ) < WINDOW_WIDTH) // OK
if (strategy == 1) // No clue what 1 means
if (strategy == SMART_MODE) // Much better
```

Even the most seemingly reasonable cosmic constant is going to change one day. You think there are 365 days per year? Your customers on Mars are going to be pretty unhappy about your silly prejudice. Make a constant so that you can easily produce a Martian version without trying to find all the 364s, 365s, and 366s in your code.

```
const unsigned DAYS_PER_YEAR = 365;
```

or

```
#define DAYS_PER_YEAR 365
```

Runtime

Newlines Terminate Output

In a program that uses console output, always terminate the last line of output with a newline, using either endl or \n.

```
  ...
  cout << "the end"; // bad
} // end of main
  ...
  cout << "the end" << endl; // good
} // end of main
```

Unsigned for Counting

Never use a signed integer for a value that cannot be negative.

```
for (int i = 0; i < number_of_items; i++) // bad
```

Array indices, item counters, and loop counters are *natural* numbers, which means they can never be negative and so are unsigned. Almost every program in this course will include <cstdint> so that you can use unsigned ints:

```
for (unsigned i = 0; i < number_of_items; i++) //good
```