Programming Standards

These standards apply to C, C++, Java as well as most other languages.

Programs will not compile if not following language syntax/definitions. But even programs that compile can be "good" or "bad" according to coding standards. These standards are put in place to ensure programs are easy to read, modify, maintain, debug.

The following sections explain the coding standards that we follow in CS1250 and 2250, and in a number of follow-up courses. These standards are mostly consistent with commercial standards.

Tabs and Indetation

Indentation is very important to properly structure statements. Indentations can be accomplished with tabs or spaces – 2 spaces being the most common standard.
Avoid mixing spaces and tabs since tabs can be rendered differently in different editors.

Blank Lines

Blank lines should be used to separate groups of related statements. For example, you can have a blank line between variable declarations and block of code, and between two blocks of code doing different things.

```
Variable_1;
Variable_2;

Statement1;
Statement2;

Statement3; // doing something different
```

Spaces and Line Breaking

Spaces improve readability, and they are allowed between any tokens/elements of a program. As a general rule, one space should be placed around all binary operators, and none after unary operators.

```
netVal = totPay - tax;      // binary = and -
netVal=totPay-tax;          // avoid, less readable
netVal = -x;                // unary -, no space between - and x
```

Statement and directive should not contain new lines unless too long to fit on single line. String literals should not be broken by new lines (there are ways to do that).

```
netVal = totPay - tax;      // good
netVal=
      totPay-
      tax;                  // very bad
```

Comments

Comments should be used to summarize what a piece of code is designed to do and why. Commenting every line with obvious descriptions of what the code does actually make the readability of the code more difficult. When the code is doing something that might not be immediately obvious to the reader, the single line comments can be used. Old-style comments are generally used as comment boxes, for example to document a function.

```
// new style one-line comment
/* old-style one-line comment */
```

Identifiers

Identifiers are used to name types, functions, variables, and data. All identifiers should use meaningful name describing the purpose without further comment. This produces self-documenting code.

To construct a meaningful name, you generally combine multiple words – the first word should start with lower case letter (except for classes), each next word with upper case or underscore.

Single-character variables should only be used for counters (i, j) or coordinates (x, y, z).

Variables are generally declared at the beginning of scope (beginning of file for global variables and beginning of function for local variables).

You should not use global variables unless explicitly allowed.

```
totalSales   // Valid
total_Sales  // Valid
total.Sales  // Invalid(Cannot contain .)
4thQtrSales  // Invalid(Cannot begin with digit)
totalSale$   // Invalid(Cannot contain $)
```

Constants

You should define named constants as `const` variables instead of symbolic constants (`#defined`) or literals (values). Constants are generally in all upper case.

```
const int ARRAYSIZE = 10;   // good, must have value
#define ARRAYSIZE 10        // worse
int array[10];              // worst
```

{} Placement

Braces should go on their own line. The closing brace should also always be at the same column as the corresponding opening brace.

```
if (condition1)
{
  // statements;
}
else
```

```
{
    statements;
}

for (i = 0; i < SIZE; i++)
{
     ...
}

while (condition1)
{
     ...
}
```

An alternative standard is to put the opening { on the same line as the statement:

```
while (condition1) {
     ...
}
```

Functions

Use functions for well-defined tasks requiring at least a few steps. In general, no function (including main) should be longer than about 20 statements. Another yardstick is – if a function cannot be printed/displayed on a single page, it is probably too long. Except for main, each function should start with a comment box explaining function, parameters, input/output, and assumptions.
Each user-defined function must have a separate prototype and a definition.

```
// prototype
float computeNetIncome(float income, float taxRate);

/* comment box for function definition */
float computeNetIncome(float income, float taxRate)
{
  return income * (1 – taxRate);
}
```

Array

The array size should be declared as a `const` variable.

```
int main()
{
  const int SIZE=10;
  int array[SIZE];
     …
  return 0;
}
```

Passing Array to Functions

Arrays should be passed using the array notation and not pointer notation. The rightmost dimension can be left empty in the function header, all other sizes must be given as literals. The missing rightmost dimension should be passed as an additional parameter unless it is known in the function otherwise.

```
// assume function prototypes, assume NUM1 and NUM2 are known
void f(int a[], const int aSize);
void g(int b[NUM1][], const int bSize);
// and assume two arrays
const int NUM1 = 5, NUM2 = 10;
int oneDim[NUM1],
int twoDim[NUM1][NUM2];
// then the following are proper calls
f(oneDim, NUM1);
g(twoDim,NUM2);
```

In the implementation of f and g functions the arguments are processed as arrays.

File

A source file has includes for the header files, function prototypes, the main function, and the other function definitions. These groups should be listed in a specific order, separated by blank lines.

```
#include .. // all includes

// macros, if any

// function prototypes for the function definitins below

// global variables if any

int main()
{     …
}

// other function definitions for the prototypes above
```

Object Oriented Naming

Class names start with an upper case letter, while members (data and methods) identifiers start with lower case letters. You should have at most one public class per file, named the same as the class. That is, every public class should be defined in a separate file.
For example, if you have a class named `ClassName`, its definition should be in file `ClassName.h` and its implementation in `ClassName.cpp`.

Attributes and Methods

Member data should usually be private, accessible only as needed with public methods.

Member methods should usually be public and so can be accessed by any class or function.

If an attribute of type `Type` is named `xyz`, it should have access methods (if desired) of:

```
Type getXyz();              // getter, read
void setXyz(Type _xyz);     // setter, write
```

Multi-File Program Structure

Anything defined in a source file that needs to be available (exported, linked) to other files of the same program needs to be prototyped in a header file.

For example, if you define class Person in file `Person.cpp` then you should provide the declarations (prototypes) in `Person.h`. The header files should be included by any file using the class Person.

```
// file Person.h
Class Person {

   // data, method prototypes, possibly inline methods here
};

// file Person.cpp
#include <iostream> // as needed
#include "Person.h"
…. // definitions of non-inline methods

// some other application file using Person
#include "Person.h"
…
   Person john;
…
```

Multiple-Inclusion Prevention

To prevent multiple inclusion (indirect) of header files, every header file should have the following:

```
// file filename.h
#ifndef FILENAME_H
#define FILENAME_H
… // the contents of the header file
#endif
```

Command-line Arguments

If a program is invoked with command-line arguments, the main function must be declared with two parameters, and they must be processed at the beginning of main:

```
int main (int argc, char* argv[]) {
  // first process arg
}
```