Quadratic Solver Programming Standard

At the top of your source code include basic information
If we don't know who you are, you won't get any points!
Identify yourself, and the program

```
/*
Jonathan Pennington //name
Quadratic Solver //project
9/7/2017 //date
*/
```

Summarize how to use your program and how it works in a brief but descriptive manner.
Points will **NOT** be taken off for this (but the better I understand your program, the higher grade you get)

```
/*
A user first reads the coding standards.
Then the user chooses to follow it or not.
If a user follows them, they receive better grades.
Otherwise they receive deductions from full credit
These standards can be found in repository a0
*/
```

Make sure to comment your code appropriately
Points will be taken off for inadequate commenting!
This means explaining lines of code that are not **instantly** clear. Anytime you do more than one operation on a single line. A comment is need.
Anytime you have a unique line of code that is not often done, and may be **confusing**. A comment is needed.
When in doubt, comment.

However, simple lines should not be commented, it adds unneeded text to your file.
(points will not be taken off for over commenting, it just wastes your time and bugs me)
Whenever appropriate use error checking (once introduced in class)
Points will be taken off for not error checking (when error checking is possible)!

Error conditions are clearly defined in Stevens. If an error condition is not defined, than no error checking is needed.

Make sure to comment your code appropriately
Points will be taken off for inadequate commenting!
This means explaining lines of code that are not instantly clear. Anytime you do more than one operation on a single line. A comment is need.

Make **brief** comments explaining what a method does before **every** method. Document inputs, basic processing purpose, and return values for all methods. Document all variables.

II) C Style and Practice

Below you will find much more specific direction regarding C programming. For example, the naming of variables, the formatting of control structures. This is our style manual. This is taken from https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html
Some of these are suggestions that are made with a good purpose in mind. We will not hunt down and penalize all violations of several of these, but your code and programming will be better if you embrace many of these suggestions

Make Names Fit
Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh!

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name that "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.
I
f you find all your names could be Thing and DoIt then you should probably revisit your design.

Function Names
• _Usually every function performs an action, so the name should make clear what it does: check_for_errors() instead of error_check(), dump_data_to_file() instead of data_file(). This will also make functions and data objects more distinguishable.

Structs are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.
• _Suffixes are sometimes useful: ○ *max* - to mean the maximum value something can have.
○ *cnt* - the current count of a running count variable.
○ *key* - key value.

For example: retry_max to mean the maximum number of retries, retry_cnt to mean the current retry count.
• _Prefixes are sometimes useful: ○ *is* - to ask a question about something. Whenever someone sees *is* they will know it's a question.
○ *get* - get a value.
○ *set* - set a value.

For example: is_hit_retry_limit.

Structure Names

• _Use underbars ('_') to separate name components

Example

```
struct foo {
struct foo *next; /* List of active foo */
struct mumble amumble; /* Comment for mumble */
int bar;
unsigned int baz:1, /* Bitfield; line up entries if desired */
fuz:5,
zap:2;
uint8_t flag;
};
struct foo *foohead; /* Head of global foo list */
```

Variable Names on the Stack (automatic or local)

• _use all lower case letters
• _use '_' as the word separator.
• _It is suggested that all lower case names be used

Justification

• _With this approach the scope of the variable is clear in the code.
• _Now all variables look different and are identifiable in the code.

Example

```
int handle_error (int error_number) {
int error= OsErr();
Time time_of_error;
ErrorProcessor error_processor;
}
```

Pointer Variables

• _place the * close to the variable name not pointer type

Example

```
char *name= NULL;
char *name, address;
```

Global Variables

- _Global variables should be prepended with a 'g_'.
- _Global variables should be avoided whenever possible.

Justification

- _It's important to know the scope of a variable.

Example

Logger g_log;
Logger *g_plog;

Global Constants

- _Global constants should be all caps with '_' separators.

Justification

It's tradition for global constants to named this way. You must be careful to not conflict with other global *#define*s and enum labels.

Example

const int A_GLOBAL_CONSTANT= 5;

#define and Macro Names

- _Put names of #defines and macros in all upper using '_' separators. Macros are capitalized, parenthesized, and should avoid side-effects. Spacing before and after the macro name may be any whitespace, though use of TABs should be consistent through a file. If they are an inline expansion of a function, the function is defined all in lowercase, the macro has the same name all in uppercase. If the macro is an expression, wrap the expression in parenthesis. If the macro is more than a single statement, use ``do { ... } while (0)'', so that a trailing semicolon works. Right-justify the backslashes; it makes it easier to read.

Justification

This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.
Some subtle errors can occur when macro names and enum labels use the same name.

Example

```
#define MAX(a,b) blah
#define IS_ERR(err) blah
#define MACRO(v, w, x, y) \
do { \
v = (x) + (y); \
w = (y) + 2; \
} while (0)
```

Formatting

Brace Placement

Of the three major brace placement strategies one is recommended:

if (condition) { while (condition) {
... ...
} }

When Braces are Needed

All if, while and do statements must either have braces or be on a single line.

Always Uses Braces Form

All if, while and do statements require braces even if there is only a single statement within the braces. For example:

if (1 == somevalue) {
somevalue = 2;
}

Justification

It ensures that when someone adds a line of code later there are already braces and they don't forget. It provides a more consistent look. This doesn't affect execution speed. It's easy to do.

One Line Form

if (1 == somevalue) somevalue = 2;

Justification

It provides safety when adding new lines while maintaining a compact readable form.

Add Comments to Closing Braces

Adding a comment to closing braces can help when you are reading code because you don't have to find the begin brace to know what is going on.

while(1) {
if (valid) {
} /* if valid */
else {
} /* not valid *
} /* end forever */

Consider Screen Size Limits

Some people like blocks to fit within a common screen size so scrolling is not necessary when reading code.

Parens () with Key Words and Functions Policy

• _Do not put parens next to keywords (such as if or while). Put a space between.
• _Do put parens next to function names.
• _Do not use parens in return statements when it's not necessary.

Justification
Keywords are not functions. By putting parens next to keywords keywords and function names are made to look alike.
Example
```
if (condition) {
}
while (condition) {
}
strcpy(s, s1);
return 1;
```
A Line Should Not Exceed 78 Characters
• _Lines should not exceed 78 characters.

Justification
• _Even though with big monitors we stretch windows wide our printers can only print so wide. And we still need to print code.
• _The wider the window the fewer windows we can have on a screen. More windows is better than wider windows.
• _We even view and print diff output correctly on all terminals and printers.

***If Then Else* Formatting**
Layout
It's up to the programmer. Different bracing styles will yield slightly different looks. One common approach is:
```
if (condition) {
} else if (condition) {
} else {
}
```
If you have *else if* statements then it is usually a good idea to always have an else block for finding unhandled cases. Maybe put a log message in the else even if there is no corrective action taken.
Condition Format
Always put the constant on the left hand side of an equality/inequality comparison. For example:
```
if ( 6 == errorNum ) ...
```
One reason is that if you leave out one of the = signs, the compiler will find the error for you. A second reason is that it puts the value you are looking for right up front where you can find it instead of buried at the end of your expression. It takes a little time to get used to this format, but then it really gets useful.
***switch* Formatting**
• _Falling through a case statement into the next case statement shall be permitted as long as a comment is included.

• _The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
• _If you need to create variables put all the code in a block.

Example

```
switch (...)
{
case 1:
...
/* comments */
case 2:
{
int v;
...
}
break;
default:
}
```

Use of *continue,break* and *?:*

Continue and Break

Continue and break are really disguised gotos so they are covered here. [PLEASE BEWARE rgt]
Continue and break like goto should be used sparingly as they are magic in code. With a simple spell the reader is beamed to god knows where for some usually undocumented reason.
The two main problems with continue are:
• _It may bypass the test condition
• _It may bypass the increment/decrement expression

Consider the following example where both problems occur:

```
while (TRUE) {
...
/* A lot of code */
...
if (/* some condition */) {
continue;
}
...
/* A lot of code */
...
if ( i++ > STOP_VALUE) break;
}
```

Note: "A lot of code" is necessary in order that the problem cannot be caught easily by the programmer.
From the above example, a further rule may be given: Mixing continue with break in the same loop is a sure way to disaster.

?:

The trouble is people usually try and stuff too much code in between the ? and :. Here are a couple of clarity rules to follow:

- _Put the condition in parens so as to set it off from other code
- _If possible, the actions for the test should be simple functions.
- _Put the action for the then and else statement on a separate line unless it can be clearly put on one line.

Example

```
(condition) ? funct1() : func2();
```

or

```
(condition)
? long statement
: another long statement;
```

One Statement Per Line

There should be only one statement per line unless the statements are very closely related.
The reasons are:
1. The code is easier to read. Use some white space too. Nothing better than to read code that is one line after another with no white space or comments.

One Variable Per Line

Related to this is always define one variable per line:
Not:

```
char **a, *x;
```

Do:

```
char **a = 0; /* add doc */
char *x = 0; /* add doc */
```

The reasons are:
1. Documentation can be added for the variable on the line.
2. It's clear that the variables are initialized.
3. Declarations are clear which reduces the probablity of declaring a pointer when you meant to declare just a char.

Use Header File Guards

Include files should protect against multiple inclusion through the use of macros that "guard" the files. Note that for C++ compatibility and interoperatibility reasons, do **not** use underscores '_' as the first or last character of a header guard (see below)

```
#ifndef sys_socket_h
#define sys_socket_h /* NOT _sys_socket_h_ */
#endif
```

Initialize all Variables

• _You shall always initialize variables. Always. Every time. gcc with the flag -W may catch operations on uninitialized variables, but it may also not.

Justification

• _More problems than you can believe are eventually traced back to a pointer or variable left uninitialized.

Short Functions

• _Functions should limit themselves to a single page of code.

Justification

• _The idea is that the each method represents a technique for achieving a single objective.
• _Most arguments of inefficiency turn out to be false in the long run.
• _True function calls are slower than not, but there needs to a thought out decision (see premature optimization).

Document Null Statements

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
{
; // <= What is (not) happening here ? Trust me.
}
```

Do Not Default If Test to Non-Zero

Do not default the test for non-zero, i.e.

```
if (FAIL != f())
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., **if (!(bufsize % sizeof(int)))** should be written instead as **if ((bufsize % sizeof(int)) == 0)** to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*.

```
#define STREQ(a, b) (strcmp((a), (b)) == 0) [N.B. Understand this ]
```

Or better yet use an inline method:

```
inline bool
string_equal(char* a, char* b)
{
(strcmp(a, b) == 0) ? return true : return false;
Or more compactly:
return (strcmp(a, b) == 0);
}
```

Note, this is just an example, you should really use the standard library string type for doing the comparison. The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

- _Returns 0 for false, nothing else.
- _Is named so that the meaning of (say) a **true** return is absolutely obvious. Call a predicate is_valid(), not check_valid().

Documentation

Comments Should Tell a Story

Consider your comments a story describing the system. Expect your comments to be extracted by a robot and formed into a man page. Class comments are one part of the story, method signature comments are another part of the story, method arguments another part, and method implementation yet another part. All these parts should weave together and inform someone else at another point of time just exactly what you did and why.

Document Decisions

Comments should document decisions. At every point where you had a choice of what to do place a comment describing which choice you made and why. Archeologists will find this the most useful information.

Comment Layout

Each part of the project has a specific comment layout. Doxygen is a document production tool that has a recommended format for the comment layouts.

Commenting function declarations

Functions prototypes should be in the file where they are declared. This means that most likely the functions will have a prototype in the .h file. However, functions like main() with no explicit prototype declaration in the .h file, should have a header in the .c file.

General advice

This section contains some miscellaneous do's and don'ts.

• _Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).
• _Compilers have bugs. Common trouble spots include structure assignment and bit fields. You cannot generally predict which bugs a compiler has. You could write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write ``around'' compiler bugs only when you are forced to use a particular buggy compiler.
• _Accidental omission of the second ``='' of the logical compare is a problem. The following is confusing and prone to error.
• _if (abool= bbool) { ... }

Does the programmer really mean assignment here? Often yes, but usually no. The solution is to just not do it, an inverse Nike philosophy. Instead use explicit tests and avoid assignment with an implicit test. The recommended form is to do the assignment before doing the test:

```
abool= bbool;
if (abool) { ... }
```

Be Const Correct

C provides the *const* key word to allow passing as parameters objects that cannot change to indicate when a method doesn't modify its object. Using const in all the right places is called "const correctness." It's hard at first, but using const really tightens up your coding style. Const correctness grows on you.

File Extensions

In short: Use the *.h* extension for header files and *.c* for source files.

No Data Definitions in Header Files

Do not put data definitions in header files. for example:

```
/*
* aheader.h
*/
int x = 0;
```

1. It's bad magic to have space consuming code silently inserted through the innocent use of header files.
2. It's not common practice to define variables in the header file so it will not occur to developers to look for this when there are problems.
3. Consider defining the variable once in a .c file and use an extern statement to reference it.

No Magic Numbers

A magic number is a bare naked number used in source code. It's magic because no-one has a clue what it means including the author inside 3 months. For example:

```
if (22 == foo) { start_thermo_nuclear_war(); }
```

```
else if (19 == foo) { refund_lotso_money(); }
else if (16 == foo) { infinite_loop(); }
else { cry_cause_im_lost(); }
```

In the above example what do 22 and 19 mean? If there was a number change or the numbers were just plain wrong how would you know?

Instead of magic numbers use a real name that means something. You can use *#define* or constants or enums as names. Which one is a design choice. For example:

```
#define PRESIDENT_WENT_CRAZY (22)
const int WE_GOOFED= 19;
enum {
THEY_DIDNT_PAY= 16
};
if (PRESIDENT_WENT_CRAZY == foo) { start_thermo_nuclear_war(); }
else if (WE_GOOFED == foo) { refund_lotso_money(); }
else if (THEY_DIDNT_PAY == foo) { infinite_loop(); }
else { happy_days_i_know_why_im_here(); }
```

Now isn't that better? The const and enum options are preferable because when debugging the debugger has enough information to display both the value and the label. The #define option just shows up as a number in the debugger which is very inconvenient. The const option has the downside of allocating memory. Only you know if this matters for your application.

Error Return Check Policy

1. Check every system call for an error return, unless you know you wish to ignore errors. For example, *printf*returns an error code but rarely would you check for its return code. In which case you can cast the return to**(void)** if you really care.
2. Include the system error text for every system error message. [N.B. We have Stevens' error.c rgt]
3. Check every call to malloc or realloc unless you know your versions of these calls do the right thing. You might want to have your own wrapper for these calls, including new, so you can do the right thing always and developers don't have to make memory checks everywhere.