Coding Practices for Programming

1. Spaces: A single blank space is represented as '□'. If not the end of the statement ';' and ',' are followed by a single space. Others (like '==', '>', '<', '=') are preceded another space. The exceptions are '&', '++' and '--'. The paranthesis need not be preceded or followed by spaces during typecasting in the inner side.

2. Indentation: A tab of size 8 characters. When you go too deep to begin the line this reminds fixing of your program!

```
#include <iostream>

int□main()
{
        std::cout□<<□"Hello\n";

        return□0;
}
```

```
#include <iostream>

int main()
{
        std::cout << "Hello\n";

        return 0;
}
```

3. Braces: For functions and methods:

```
int□function(□int□x□)
{
        (body of function)
}
```

```
int function( int x )
{
        (body of function)
}
```

The following examples are for all other cases, and should be applied even if braces can be done away with as in if, while, etc.

```
while□(□true□)□{
        for (□int□i□=□0; □i□<□10;□i++□) {
                if□(□i□!=□5□)□{
                        cout□<<□"Yes\n";
                }□else□if□(□i□!=□6□)□{
                        cout□<<□"No\n";
                }□else□{
                        cout□<<□"Done\n";
                }
        }
}
```

```
while ( true ) {
        for ( int i = 0; i < 10; i++ ) {
                if ( i != 5 ) {
                        cout << "Yes\n";
                } else if ( i != 6 ) {
                        cout << "No\n";
                } else {
                        cout << "Done\n";
                }
        }
}
```

4. Classes: If you wish to inline the following is the format.

```
class□C□{
        private:
                int□z;

        public:
                int□x;
                int□y;

                C()□{}

                prt(□int□_x□)
                {
                        cout□<<□_x□<<□endl;
                }
};
```

```
class C {
        private:
                int z;

        public:
                int x;
                int y;

                C() {}

                prt( int _x )
                {
                        cout << _x << endl;
                }
};
```

5. Comments: *Do not over-comment.* Never try to explain how your code works in a comment: it's much better to write the code so that the working is obvious. Generally, you want your comments to tell what your code does, not how. You can make small comments to note or warn about something particularly clever (or ugly), but try to avoid excess. Instead, put the comments at the *head* of the function, telling people what it does, and *only if necessary*, how it does so.

An important comment
```
void swap( int &a, int &b )
{
       /*
        * swap variables
        */
       a ^= b ^= a ^= b;
}
```

A small comment
```
void swap( int &a, int &b )
{
       // swap variables
       a ^= b ^= a ^= b;
}
```

6. **Functions and methods**

Functions should be short and sweet, and do just one thing. They should fit on one or two screenfuls of text (the ISO/ANSI screen size is 80x24, as we all know), and do one thing and do that well.

The maximum length of a function is inversely proportional to the complexity and indentation level of that function. So, if you have a conceptually simple function that is just one long (but simple) case-statement, where you have to do lots of small things for a lot of different cases, it's ok to have a longer function.

However, if you have a complex function, you should adhere to the maximum limits all the more closely. Use helper functions with descriptive names (you can the compiler to in-line them if you think it's performance-critical).

Another measure of the function is the number of local variables. They shouldn't exceed 5-10, or you're doing something wrong. Re-think the function, and split it into smaller pieces. A human brain can generally easily keep track of about 7 different things, anything more and it gets confused.

```
void add( int i, int j )
{
       cout << "Sum is: " << i + j << endl;
}
```

7. Containers and iterators

```
vector<int> v;
vector<int>::iterator it_v;
map< pair<string, int> m;
vector< vector<int> > vv;
list< vector<int> >::iterator it_vv;
```

8. Naming: Mixed-case names (like ThisVariableIsATemporaryCounter) are frowned upon, descriptive names for global variables are a must. To call a global function "foo" is a shooting offense.

Global variables (to be used only if you really need them) need to have descriptive names, as do global functions. If you have a function that counts the number of active users, you should call that "count_active_users()" or similar, you should not call it "cntusr()".

Local variable names should be short, and to the point. If you have some random integer loop counter, it should probably be called "i". Calling it "loop_counter" is non-productive, if there is no chance of it being misunderstood. Similarly, "tmp" can be just about any type of variable that is used to hold a temporary value.

9. `typedef` : `typedef` should not be used unless it is really required, probably to shorten huge data types

```
typedef vector< vector<int> > vec_int;
typedef vector< vector<int> >::iterator it_vvec_int;
typedef vector< vector<string> > vv_string;
```

10. Pointers and references: Pointers and references must be next to the variable names.

```
void ( int *x, int *y, int &z );
```

Sample Program

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector< int > vec_int;

void add( vec_int &vx, vec_int &vy, vec_int &vz )
{
        // insert vx and vy into vz

        for ( int i = 0; i < (int) vx.size(); i++ ) {
                vz.push_back( vx[i] );
        }

        for ( int i = 0; i < (int) vy.size(); i++ ) {
                vz.push_back( vy[i] );
        }
}

int main()
{
        vec_int v1;
        vec_int v2;
        vec_int v3;

        for ( int i = 0; i < 10; i++ ) {
                v1.push_back( i );
        }

        for ( int i = 20; i < 30; i++ ) {
                v2.push_back( i );
        }

        add( v1, v2, v3 );

        for ( int i = 0; i < (int) v3.size(); i++ ) {
                cout << v3[i] << endl;
        }

        return 0;
}
```