CS 162C++
TTT Suggestions

Here are some things that might help you with the TTT problem.

Starting the program

When you are given a problem like this to solve, many people immediately jump into their IDE and start writing code. A better approach is to sit down and think about how the program should function and then build it incrementally doing one step at a time.

For example, the first step here might be to look back at a similar problem you did last term – the game of Nim. Look at the basic structure of the game and then create a program framework using comments to lay out how you want to tackle it.

```
#include <iostream>

using namespace std;

// function declarations

int main()
{
         // outer repeat loop
    do {

        // display instruction

        // set up game

            // inner game loop
        do {

            // display board

            // get next move

            // update board

            // check for win

            // check for tie

            // if game not over, swap player

        } while (not a win or a tie);

        // display final board and congratulate winner

    } while ( not repeatGame );

    return 0;
}
```

Now, think of what functions you want to use and declare them. For the first pass, any that you are not sure of, you can leave as void return type with no parameters. Add those details later as you find how the function will be used.

```
#include <iostream>
using namespace std;

// global constants
const int ROW = 3;
const int COL = 3;
const char X = 'X';
const char O = 'O';
const char SPACE = ' ';

// function declarations
void showInstructions();
void showBoard();
void initBoard();
void getMove();
void updateBoard();
bool checkWin();
bool checkTie();
bool repeat();

int main()
{
    // outer repeat loop
    do {
        // display instructions
        showInstructions();

        // set up game and define variables
        bool win, tie;
        char player = X;
        char theBoard[ROW][COL];
        initBoard();

            // inner game loop
        do {
            // display board
            showBoard();

            // get next move
            getMove();

            // update board
            updateBoard();

            // check for win
            win = checkWin();

            // check for tie
            tie = checkTie();

            // if game not over, swap player
            if ( not win and not tie )
            {
                // swap player
            }
        } while (not win and not tie);

        // display final board and congratulate winner
        showBoard();
        if ( win )
            cout << "Congratulations " << player << " you won." << endl;
        if ( tie )
            cout << "The game was a tie." << endl;
    } while ( repeat() );
    return 0;
}

// function definitions
void showInstructions()
{
    cout << "Showing instructions" << endl;
}
void showBoard()
{
    cout << "Showing board" << endl;
}
void initBoard()
{
    cout << "Initializing board" << endl;
}
void getMove()
{
    cout << "Getting a move" << endl;
}
void updateBoard()
{
    cout << "Updating board" << endl;
}
bool checkWin()
{
    cout << "Check for win" << endl;
    return false;
}
bool checkTie()
{
    cout << "Check for tie" << endl;
    return true;
}
bool repeat()
{
    cout << "checking for repeat" << endl;
    return false;
}
```

Now you can continue to develop the program one function at a time. Each time make sure that it compiles and runs without problems.

I did it in this order:

1. display instructions
2. initializeBoard – or initialize it in main
3. displayBoard
4. getMove
5. checkWin
6. checkTie (no win and no spaces or no win and nine moves)
7. repeat

By using this order, each step built upon the previous and it kept the game compiling and showing the latest change.

Displaying the board

There are two different approaches to displaying the board. One uses pairs of [] to surround each space. The other puts out | between each column and ---------- between each row.

The two displays look something like this:

```
     [X][ ][ ]
     [ ][O][ ]
     [ ][ ][ ]
```

The other looks like this:

```
   1   2   3
1  X |   |
-------------
2    | O |
-------------
3    |   |
```

The first you can do by adding [and] when you output the contents of a board location. The second requires that you add some code to your display function to add the appropriate extra characters at the right time. Either is fine. Remember that the way to show the board is with nested loops – the outer loop is for rows and the inner loop is for columns within a row.

Defining the board

There are two different approaches to creating your board. You can use a 1D array of 9 elements or you can use a 2D array of 3x3 elements. Either would work fine. The second is easier to pass to functions, but the first is easier to design and test your program.

Passing the board

Rather than having a global variable for the board, you should define it in main and pass it to functions as necessary. If you have a one-dimensional board you could do something like this:

```
void display(char theBoard[], int size);
```

If you have a two-dimensional board, you need to go with:

```
void display(char theBoard[][COL], int row, int col);
```

Getting and validating input

If you are using a two dimensional array, you will need to ask the user for the row and column where they want to move. You need to have a while loop that verifies that the row and column are both within bounds (0-2). You also need to verify that they are moving to an empty space – if (board[row][col] == SPACE). If it is not an empty space, then you should loop back and get a new row and column and re-validate.

```
do {
    if ( row or column is out of bounds or if input bad)
        display message and clear buffer
    else if ( the space is used )
        display message and clear buffer
} while ( out of bounds or not a clear space );
```

When you are looking to input an integer, one way to see if the user entered an integer or something else is to use the function cin.fail(). This will return true if you were trying to input a integer and got something else.

```
cin >> input;

if ( cin.fail() )
    bad input, do something
```

Another way is to have your read inside an if statement

```
if ( not cin >> input )
   bad input, do something
```

When you have bad input, you need to clear the input flags and then ignore anything that is currently in the buffer.

```
cin.clear();
cin.ignore(numeric_limits<streamsize>::max(), ‘\n’);
```