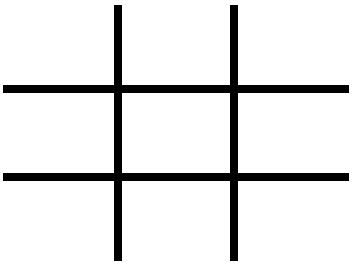
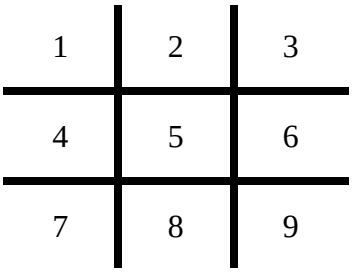
CS102: Tic-Tac-Toe

The game of **Tic-Tac-Toe** is a two-player game played on a 3 by 3 game board. It is customary to draw the board without showing the outer border. For example:



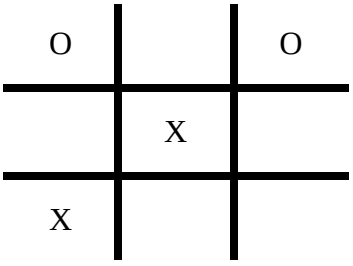
Tic-Tac-Toe board

We are going to number the squares as follows:



Numbered Tic-Tac-Toe board

One player is the X player and the other player is the O player. Players alternate turns. When it is the X player's turn, that player can draw an X in any empty square. When it is the O player's turn, that player can draw an O in any empty square. The X player always goes first. So, after each player has taken two turns, the board may look like this, and it would be the X player's turn next:



After four turns

If, after either player's turn, that player has his symbol occupy all three squares of any row, column, or main diagonal of the game board, then that player wins and the game is over. If nine turns are taken and the entire board is filled without either player winning,

the game is a draw. It is well known that if both players play perfectly, the game will result in a draw. This is clearly demonstrated in the movie "War Games".

We are going to examine a program that plays Tic-Tac-Toe against the user. The program will use the following non-perfect (i.e., beatable) strategy when deciding its next move:

1) If it can win the game with its current move, it will take the win.
2) Otherwise, if it can block the opponent from winning on the opponent's next move, it will block the opponent.
3) Otherwise, if the middle square is open, it will take the middle square.
4) Otherwise, if there is an available corner, it will take one available corner.
5) Otherwise, it will take any open square.

When it is the user's turn, the computer will ask the user to choose a square. The user will be expected to enter one integer representing the user's chosen square. We will assume that the user will correctly enter an integer (as opposed to a floating point number or text), but we will not assume that the square will be valid. If the number is out of range (less than 1 or greater than 9), or if the chosen square is not empty, we will prompt the user to enter another square. We will continue this until a valid square is entered.

The computer will ask if the user would like to go first. We will instruct the user to enter a 'y' or 'Y' if the user wants to go first, and an 'n' or 'N' if the user wants the computer to go first. Whoever goes first is the X player.

The computer will announce when it has won the game, and it will congratulate the user if the user wins. If the board is filled without anyone winning, the computer will declare the game to be a draw. At the end of a game, the computer will ask if the user wants to play again.

The computer will display the board using ASCII characters after each move by the computer or the user. For example, the board after the four turns represented above would be displayed as follows:

```
   *   *
 O *   * O
   *   *
***********
   *   *
   * X *
   *   *
***********
   *   *
 X *   *
   *   *
```

Text representation of board

After each of the computer's turns, it will output its move (designated by the number of its chosen square) before drawing the board.

The program uses 3 global variables (declared at the top of the program and shared by all its functions). One stores the current state of the Tic-Tac-Toe board, one stores the computer's symbol (either X or O depending on who goes first), and one stores the user's symbol (the opposite of the computer's symbol). The three globals are declared as follows:

```
char board[3][3];
char computer, user;
```

The program uses a 3 by 3 two-dimensional array of characters called "board" to represent the board, and each slot is filled with an 'X' or an 'O' if the corresponding square has already been taken, or a ' ' (single space) if the corresponding square is still empty.

The variables "computer" and "user" represent the two players' symbols; one will be an 'X' and one will be an 'O' depending on who goes first. (Really only one of these variables is necessary, since one can be determined from the other, but I included both in this program for simplicity throughout the code.)

The program consists of 13 functions, not including "main". The prototype statements for the functions are:

```
void init_board(void);
```
This function initializes the board by filling it in with spaces. (Because "board" is a global, it does not have to be passed to the function.)

```
void draw_board(void);
```
This function displays the current state of the board to standard output.

```
int user_first(void);
```
This function asks if the user wants to go first. Returns 1 if yes, 0 if no.

```
int play_again(void);
```
This function asks if the user wants to play again. Returns 1 if yes, 0 if no.

```
int symbol_won(char);
```
This function takes a symbol (either 'X' or 'O') as a parameter, and checks the board to determine if that symbol has already won the game. Returns 1 if yes, 0 if no.

```
int find_win(char);
```
This function takes a symbol (either 'X' or 'O') as a parameter, and checks if there is any empty square such that placing the symbol there would result in a win. If so, it returns the number of this square (squares are numbered 1 through 9 as indicated); otherwise, it returns 0. (I there are multiple possible wins, it returns just one of them.)

```
int middle_open(void);
```
This function returns a 5 (the number of the middle square) if the middle square is empty, or 0 otherwise.

```
int find_corner(void);
```
If there are any empty corner squares (squares 1, 3, 7, and 9), this function returns the number of one of them; otherwise, it returns 0.

```
int find_side(void);
```
If there are any empty side squares (squares 2, 4, 6, and 8), this function returns the number of one of them; otherwise, it returns 0.

```
void computer_move(void);
```
This function examines the current state of the board and implements the specified strategy rules. The first rule that applies determines the computer's next move. The move is then displayed to standard output, and the board is updated by placing the computer's symbol (either X or O) into the appropriate slot of the array representing the board. This function relies on the functions "find_win", "middle_open", "find_corner", and "find_side".

```
int square_valid(int);
```
This function takes as a parameter the number of a square and checks if it is valid (a number from 1 to 9), and if so, if the square is empty. If the square is valid and empty, it returns 1 (true), and if not, it returns 0 (false).

```
void player_move(void);
```
This function asks the user to enter a move (an integer from 1 to 9 representing a square). It continues to prompt the user until the user enters a valid, empty square (determined by "square_valid"). After the user enters an appropriate square, the function updates the board.

```
void play_game(void);
```
This function loops through the 9 turns of a game. For each turn, depending on whether the number of the turn is even or odd and who goes first, "computer_move" or "player_move" is called. After each turn, "draw_board" is called to display the board, and "symbol_won" is called to determine if either player has won. If all 9 turns go by without a winner, the game is a draw.

Of course, there is also the function "main". Here it is:

```
/*
 * Initialize the board, ask who goes first, play a game,
 * ask if user wants to play again.
 */
int main(void)
{
  while(1)
```

```
  {
    init_board();
    if (user_first())
    {
      computer = 'O';
      user = 'X';
    }
    else
    {
      computer = 'X';
      user = 'O';
    }
    play_game();
    if (!play_again())
      break;
  }
  return 0;
}
```

So, for each iteration of the loop, the board is initialized, the user decides who goes first (and the globals storing the computer's and user's symbols are assigned values accordingly), the game is played, and the user decides if there will be another game (if not, we break out of the loop and the program ends).

You might be surprised at how small this function "main" is for an approximately 300 line program, but that is actually common. The "main" function doesn't usually do much by itself when proper structured programming is implemented.

Next we will look at the functions that are simple in that they don't rely on other functions:

```
/* Make sure board starts off empty. */
void init_board(void)
{
  int row, col;

  for (row = 0; row < 3; row++)
    for (col = 0; col < 3; col++)
      board[row][col] = ' ';

  return;
}
```

This function simply uses a nested loop to traverse all slots of the two-dimensional array "board" and initialize them to spaces.

```
/* Display the board to standard output. */
```

```
void draw_board(void)
{
  int row, col;

  printf("\n");
  for (row = 0; row < 3; row++)
  {
    printf("   *   *   \n");
    printf(" %c * %c * %c \n",
        board[row][0], board[row][1], board[row][2]);
    printf("   *   *   \n");
    if (row != 2)
      printf("***********\n");
  }
  printf("\n");

  return;
}
```

This function displays the board (as demonstrated in an example earlier). Each iteration of the "for" loop is responsible for printing one row of the Tic-Tac-Toe board, which actually consists of three rows of ASCII characters. The middle of the three rows of ASCII characters includes three characters indicating which symbols, if any, are in the corresponding squares. Since we have decided to represent X's and O's with the characters 'X' and 'O' in "board" and empty squares with spaces, we can print these characters directly.

```
/*
 * Ask if user wants to go first.
 * Returns 1 if yes, 0 if no.
 */
int user_first(void)
{
  char response;

  printf("Do you want to go first? (y/n) ");
  do
  {
    response = getchar();
  } while ((response != 'y') && (response != 'Y') &&
       (response != 'n') && (response != 'N'));

  if ((response == 'y') || (response == 'Y'))
    return 1;
  else
    return 0;
}
```

```
/*
 * Ask if user wants to play again.
 * Returns 1 if yes, 0 if no.
 */
int play_again(void)
{
  char response;

  printf("Do you want to play again? (y/n) ");
  do
  {
    response = getchar();
  } while ((response != 'y') && (response != 'Y') &&
       (response != 'n') && (response != 'N'));

  if ((response == 'y') || (response == 'Y'))
    return 1;
  else
    return 0;
}
```

These functions ask the user a question and loop until a 'y', 'Y', 'n', or 'N' is encountered. Even if it were certain the user would type just one letter, these loops would still be necessary in case something was left over in the buffer before the loop was encountered.

```
/*
 * If middle square is empty, return 5;
 * otherwise return 0.
 */
int middle_open(void)
{
  if (board[1][1] == ' ')
    return 5;
  else
    return 0;
}

/*
 * Return the number of an empty corner, if one exists;
 * otherwise return 0.
 */
int find_corner(void)
{
  if (board[0][0] == ' ')
    return 1;
  if (board[0][2] == ' ')
```

```
    return 3;
  if (board[2][0] == ' ')
    return 7;
  if (board[2][2] == ' ')
    return 9;

  return 0;
}

/*
 * Return the number of an empty corner,
 * if one exists; otherwise return 0.
 */
int find_side(void)
{
  if (board[0][1] == ' ')
    return 2;
  if (board[1][0] == ' ')
    return 4;
  if (board[1][2] == ' ')
    return 6;
  if (board[2][1] == ' ')
    return 8;

  return 0;
}
```

Above are very simple functions that use "if" statements to perform their tasks.

```
/* Check if the given square is valid and empty. */
int square_valid(int square)
{
  int row, col;

  row = (square - 1) / 3;
  col = (square - 1) % 3;

  if ((square >= 1) && (square <= 9))
  {
    if (board[row][col] == ' ')
      return 1;
  }

  return 0;
}
```

This function checks if the passed "square", as entered by a user, represents a valid square. The two assignment statements at the beginning convert "square" (i.e., a number from 1 to 9 representing a spot on the board as described earlier) to the appropriate row and column (each in the range 0 to 2). If that spot is empty, 1 (TRUE) is returned; otherwise 0 (FALSE) is returned.

```
/* Check if the given symbol has already won the game. */
int symbol_won(char symbol)
{
  int row, col;

  for (row = 0; row < 3; row++)
  {
    if ((board[row][0] == symbol) &&
     (board[row][1] == symbol) &&
     (board[row][2] == symbol))
      return 1;
  }

  for (col = 0; col < 3; col++)
  {
    if ((board[0][col] == symbol) &&
     (board[1][col] == symbol) &&
     (board[2][col] == symbol))
      return 1;
  }

  if ((board[0][0] == symbol) &&
      (board[1][1] == symbol) &&
      (board[2][2] == symbol))
    return 1;

  if ((board[0][2] == symbol) &&
      (board[1][1] == symbol) &&
      (board[2][0] == symbol))
    return 1;

  return 0;
}
```

The first "for" loop checks the rows, the second "for" loop checks the columns, and the remaining two "if" statements check the diagonals to determine if the passed symbol has already won the game.

Using these functions that have already been described, the four remaining functions can be implemented. For example:

```
/*
 * Find a win, if any, for the given symbol.
 * If a winning square exists, return the square;
 * otherwise, return 0.
 */
int find_win(char symbol)
{
  int square, row, col;
  int result = 0;

  /*
   * Loop through the 9 squares.
   * For each, if it is empty, fill it in with the given
   * symbol and check if this has resulted in a win.
   * If so, keep track of this square in result.
   * Either way, reset the square to empty afterwards.
   * After the loop, if one or more wins have been found,
   * the last will be recorded in result.
   * Otherwise, result will still be 0.
   */
  for (square = 1; square <= 9; square++)
  {
    row = (square - 1) / 3;
    col = (square - 1) % 3;

    if (board[row][col] == ' ')
    {
      board[row][col] = symbol;
      if (symbol_won(symbol))
        result = square;
      board[row][col] = ' ';
    }
  }

  return result;
}
```

Here we have used the fact that "symbol_won" already exists to code "find_win" without many lines of code. The comment explains the logic of the function. Each possible square is converted to a row and column, and if placing the passed "symbol" in any of the squares would result in a win, the last such square will be returned to the caller. Another alternative method for writing this function would have been to have 9 complex "if" statements to determine if any of the 9 possible squares could results in a win. That might have looked like this:

```
/*
 * Find a win, if any, for the given symbol.
 * If a winning square exists, return the square;
```

```
 * otherwise, return 0.
 */
int find_win(char symbol)
{
  if ((board[0][0] == ' ') &&
      (((board[0][1] == symbol) && (board[0][2] == symbol)) ||
       ((board[1][0] == symbol) && (board[2][0] == symbol)) ||
       ((board[1][1] == symbol) && (board[2][2] == symbol))))
    return 1;

  if ((board[0][1] == ' ') &&
      (((board[0][0] == symbol) && (board[0][2] == symbol)) ||
       ((board[1][1] == symbol) && (board[2][1] == symbol))))
    return 2;

  if ((board[0][2] == ' ') &&
      (((board[0][0] == symbol) && (board[0][1] == symbol)) ||
       ((board[1][2] == symbol) && (board[2][2] == symbol)) ||
       ((board[1][1] == symbol) && (board[2][0] == symbol))))
    return 3;

  if ((board[1][0] == ' ') &&
      (((board[0][0] == symbol) && (board[2][0] == symbol)) ||
       ((board[1][1] == symbol) && (board[1][2] == symbol))))
    return 4;

  if ((board[1][1] == ' ') &&
      (((board[0][0] == symbol) && (board[2][2] == symbol)) ||
       ((board[0][2] == symbol) && (board[2][0] == symbol)) ||
       ((board[0][1] == symbol) && (board[2][1] == symbol)) ||
       ((board[1][0] == symbol) && (board[1][2] == symbol))))
    return 5;

  if ((board[1][2] == ' ') &&
      (((board[0][2] == symbol) && (board[2][2] == symbol)) ||
       ((board[1][1] == symbol) && (board[1][0] == symbol))))
    return 6;

  if ((board[2][0] == ' ') &&
      (((board[2][1] == symbol) && (board[2][2] == symbol)) ||
       ((board[0][0] == symbol) && (board[1][0] == symbol)) ||
       ((board[1][1] == symbol) && (board[0][2] == symbol))))
    return 7;

  if ((board[2][1] == ' ') &&
      (((board[2][0] == symbol) && (board[2][2] == symbol)) ||
       ((board[1][1] == symbol) && (board[0][1] == symbol))))
    return 8;

  if ((board[2][2] == ' ') &&
      (((board[2][0] == symbol) && (board[2][1] == symbol)) ||
       ((board[0][2] == symbol) && (board[1][2] == symbol)) ||
       ((board[1][1] == symbol) && (board[0][0] == symbol))))
    return 9;

  return 0;
}
```

This is possibly more efficient computationally, but as you can see, it takes much more code, and it is very easy to make a careless mistake that might be hard to track down when dealing with code like this.

Now that "find_win", "middle_open", "find_corner", and "find_side" already exist, the task of writing "computer_move", which needs to follow the strategy rules specified earlier, becomes simple:

```
/* Choose a move for the computer. */
void computer_move(void)
{
  int square;
  int row, col;

  /* Use first strategy rule that returns valid square */
  square = find_win(computer);
  if (!square)
    square = find_win(user);
  if (!square)
    square = middle_open();
  if (!square)
    square = find_corner();
  if (!square)
    square = find_side();

  printf("\nI am choosing square %d!\n", square);

  row = (square - 1) / 3;
  col = (square - 1) % 3;

  board[row][col] = computer;

  return;
}
```

Note the expression "if (!square)", which is used repeatedly. Remember that the functions which "computer_move" is calling return 0 when they don't find a win or a square of the correct type. A 0 is FALSE when treated as a Boolean expression. So as long as "square" remains 0, it means no square was found using the strategy rules checked so far, and we check the next one. By the time we get to the "printf", some square will have been found. (If the board were full, the program would have already declared the game to be a draw.) Toward the end of the function we are converting a square to a row and a column, after which we fill in the appropriate slot of the board with the appropriate symbol (either 'X' or 'O', as stored in the global "computer").

Here is the function "player_move" which relies on "square_valid":

```
/* Have the user choose a move. */
void player_move(void)
{
  int square;
  int row, col;

  do
  {
    printf("Enter a square: ");
    scanf("%d", &square);
  } while (!square_valid(square));

  row = (square - 1) / 3;
  col = (square - 1) % 3;

  board[row][col] = user;

  return;
}
```

This function is using a "do…while" loop to repeatedly ask the user to enter a square until one is entered that is valid and empty. It is converted to a row and column, and this time the user's symbol (stored in the global "user") is stored in the appropriate slot of the board.

Now that "computer_move", "player_move", "draw_board", and "symbol_won" already exist, the "play_game" function can be written as follows:

```
/* Loop through 9 turns or until somebody wins. */
void play_game(void)
{
  int turn;

  for (turn = 1; turn <= 9; turn++)
  {
    /* Check if turn is even or odd
       to determine which player should move. */
    if (turn % 2 == 1)
    {
      if (computer == 'X')
        computer_move();
      else
        player_move();
    }
    else
    {
```

```
      if (computer == 'O')
        computer_move();
      else
        player_move();
    }

    draw_board();

    if (symbol_won(computer)) {
      printf("\nI WIN!!!\n\n");
      return;
    }
    else if (symbol_won(user)) {
      printf("\nCongratulations, you win!\n\n");
      return;
    }
  }

  printf("\nThe game is a draw.\n\n");
  return;
}
```

This function uses a "for" loop to loop through the 9 possible turns. For each turn, depending on whether the turn is odd or even and who went first, there is a single call to either "computer_move" or "player_move". After each turn, the board is displayed with a call to "draw_board", and "symbol_won" is used to determine if either player has won the game. If either the player has won, an appropriate message is displayed and a "return" statement will return control to the function that called "play_game". The only way we complete the "for" loop and get to the code after it is if all 9 turns go by without a winner, in which case the game must be a draw.

We've already seen "main", which is simple since "init_board", "user_first", "play_game", and "play_again" already exist!

Writing this Tic-Tac-Toe program without the use of functions (i.e. putting all code in "main") would be a coding nightmare. The function would be huge and unreadable. Much code would need to be repeated, and proper indentation would go off the side of the screen. Even keeping track of all the variables would be difficult. However, with proper top-down design and structured programming, the task has been divided into functions, each of which is simple to code.