Ad Hoc Problems

- "Ad hoc": It generally signifies a solution designed for a specific problem or task, non-generalizable, and not intended to be able to be adapted to other purposes (Wikipedia).
- Some are simple, some are tricky.
- Still, there are common techniques that may be used to simplify your code or the coding process.
- They are good for testing your basic programming skills with relatively little algorithmic challenges.
- The goal is to solve the problem as simply and as quickly as possible.

General Tips

- Use subroutines. For most but the simplest problems, use a subroutine to handle each case, and have the main program call it repeatedly:

```
int main()
{
  int case_num = 1;
  while (do_case(case_num++))
    ;
  return 0;
}
```

  (or read the number of cases and use a for loop.)

- It is okay to use global variables to avoid passing too many things around (just be careful).

- Avoid dynamic memory as much as possible (easy to get wrong, efficiency concerns). Use the STL if you need to.

Useful Libraries and Routines

- Being familiar with standard libraries makes life a lot easier.
- Other useful libraries:
  - Date class
  - Base conversion
  - Infix expression evaluation
  - Roman numeral conversion

Reading Problem (again)

- Many ad hoc problems will describe some situation (e.g. games, simulation) with **many** rules.
- Each rule is typically easy to implement on its own.
- It is important to understand how they interact with each other.
- And don't forget about any of them.

Data Structures/Representation

- How you choose to store or represent input data can have a dramatic impact on the complexity of your program.
- This applies also to data that are given to you in the problem statement (e.g. the denominations of the Canadian coin system), or "common" knowledge (e.g. number of days in each month).
- Think carefully about the data representations. Much will come from experience (and the following slides).

Table-driven Code

- By far, the most useful technique in simplifying the logic of your code is table-driven code.
- We will often extend the notion of "tables" to include strings as well (1D table of characters).
- Sometimes the entries are stored in an array, and both the index and the content of each element have special meanings.
- Sometimes the index is irrelevant.
- Sometimes you can store the entries in an STL map.
- The resulting code may be slightly slower, but it's usually much cleaner and easier to verify.

Example

- Check if a character is a vowel:

```
if (ch == 'A' || ch == 'E' || ... )
```

  You need 10 cases (5 if you use `toupper`).

- You can also use a `switch` statement.

- Or this:

```
const string vowels = "AEIOUaeiou";
if (vowels.find(ch) != string::npos)
```

- With many branches in `if` or `switch` statements, it is easy to forget one case. In a table, all cases are listed close together.

Example: WERTYU (10082)

- Standard QWERTY keyboard, but the keys are shifted one spot to the right.
- Given mistyped input, produce the correct input.
- You can use many cases in `if` or `switch` statements.
- Or you can quickly type an array of four strings (one string also work), search for the input character, and print the previous one.

Example: Date class

- Given year and month, return the number of days in the month.
- Write a function that returns 0 for non-leap year and 1 for leap year.
- Use a 2D array:

```
const int dayInMonth[2][13] = {
  { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
  { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

cout << dayInMonth[isLeap(year)][month] << endl;
```

Example: searching in a grid

- Often you are given a coordinate (r, c) in a grid, and you want to look at the neighbours (4 or 8).
- Bad: write code for $(r-1, c)$, then repeat it for $(r+1, c)$, $(r, c-1)$, $(r, c+1)$. Even worse for 8 directions.
- You can instead use a "delta" table:

```
const int dr[4] = {-1, 1, 0, 0};
const int dc[4] = { 0, 0, -1, 1};

for (int dir = 0; dir < 4; dir++) {
  int nr = r + dr[dir], nc = c + dc[dir];
  // do something with (nr, nc)
}
```

- Aside: don't confuse (r, c) and (x, y) coordinate systems.

Example: 8 queens problem

- Put 8 queens on a chessboard so they do not attack each other. Can be generalized to $n \times n$ boards.
- If we are given the locations of n queens, we can check to see if they attack each other by checking all pairs in $O(n^2)$ time (how?).
- We can simplify this by representing all rows, columns, diagonals, anti-diagonals with a boolean variable (whether any queen is in that line) and check in $O(n)$ time.