# AN INTRODUCTION TO PROGRAMMING

## THROUGH C++

*with*

Manoj Prabhakaran

## Lecture 22

## Revision

### Example: Backtracking

Based on material developed by Prof. Abhiram G. Ranade

# Recap

- Several programming concepts covered

| Data | Control Flow/Dynamics | Program Organization |
|---|---|---|
| Variables, expressions | Sequential execution | Statements, scope |
| Basic data types | (And sequence points) | `main()` and other functions |
| Internal representation | Conditional execution | Preprocessing |
| Reference variables | Conditional loops | Header files, Multiple C++ files |
| Structs | Function calls | Functions inside structs |
| Arrays | Lifetime of a variable | Function templates |
| From the Standard Library:<br>I/O streams, `string` | Static variables | Namespaces |
| Pointers | Recursion | Classes (a glimpse) |
| More from the Standard Library | Exception handling | |

- And not covered: Inheritance, variadic arguments, function pointers, void*, anonymous functions, concurrency (multiple threads), system calls, network programming (sockets), …

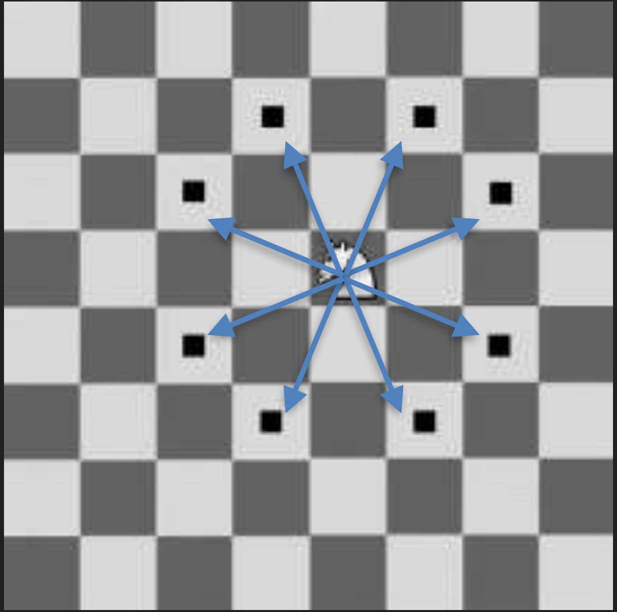
# Today

- We have seen a few algorithmic ideas along the way
  - E.g., Divide and conquer
  - E.g., memo-ization when using recursion
  - E.g., use of data structures, like stacks (e.g., RPN calculator)
- Several (very clever) algorithms that solve seemingly "intractable" problems
- But some problems don't seem to have any such algorithms
- Will need to resort to "brute-force" (if the problem is not too big)
- Today: Backtracking as a means of systematically exploring all possibilities

# Knight's Tour

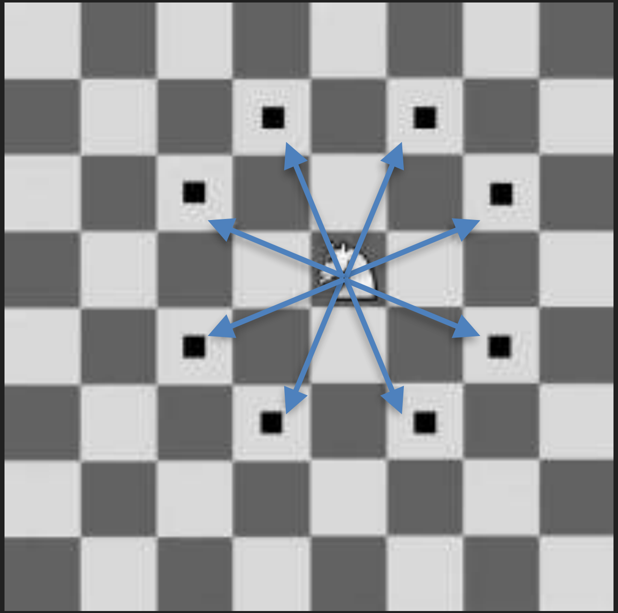
Demo

- A knight on a chessboard can make L-shaped moves: i.e., to a square 2 positions vertically and 1 position horizontally away, or 2 positions horizontally and 1 position vertically away
- Problem: Given a starting point for the knight, find 63 moves such that the knight reaches every square on the board
  - Many solutions are known
  - Our plan: let the computer search for it by brute-force
  - After finding the solution, animate it



# Steps in Solving

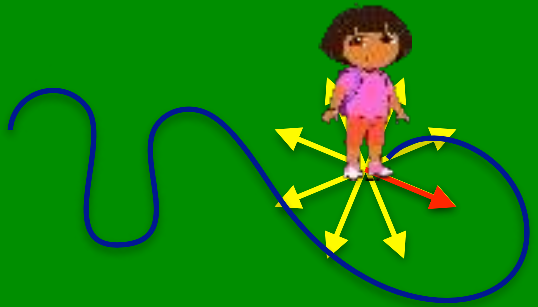
- Algorithmic idea: How to do the brute-force search
  - Recursive formulation
  - Explicitly using a stack
- C++ design
  - What classes to design, what to use from the standard library
- Simplecpp graphics
  - Will write a separate program to animate



# Backtracking

- While have not reached the goal (or backtracked past start)
    - Keep moving by arbitrarily picking the next move from available moves  (picked move becomes unavailable)
    - If we get stuck (i.e., no valid move available from here), then return to the previous position (backtrack)

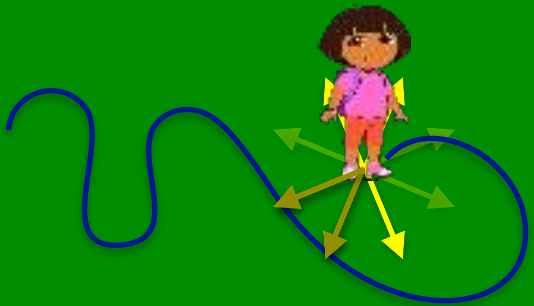
```
explorer Dora(start_coordinates);
while(!Dora.finished_exploring()) {
  if(Dora.stuck())
    Dora.backtrack();
  else
    Dora.proceed();
}
```



# Backtracking

- What should the explorer remember?
  - On reaching any location (possibly on backtracking), should know where to go next
    - – A queue of *neighbouring* locations (not already tried)
    - – A list of all locations already in the current tour

```
explorer Dora(start_coordinates);
while(!Dora.finished_exploring()) {
  if(Dora.stuck())
    Dora.backtrack();
  else
    Dora.proceed();
}
```



# A Class for the Explorer's State

`class` here works same as `typename`

```
template<class C>        // C is the class for "coordinates"
class state {
  C here;                // coordinates of the location
  vector<C> whereto;     // locations remaining to be explored
public:
  state(const C& coords) : here(coords), whereto (here.reachable()) {}
  bool stuck() { return whereto.empty(); }
  C where() { return here; }
  C next() { C x = whereto.back(); whereto.pop_back(); return x;}
};
```

class C should have this function

Removes one location from the whereto list, and returns it

# A Class for the Explorer

```
template <class C, class Hash>   //C for coordinates, Hash used for hashing C
class explorer {
  vector<state<C>> path;         //path so far
  unordered_set<C,Hash> visited; //set of locations in path (for quickly checking)
public:
  explorer(C start) {
    path.push_back({start});     //shorthand for path.push_back(state<C>(start));
    visited.insert(start);
  }
  bool stuck() { return path.back().stuck(); }
  void backtrack() { visited.erase(path.back().where()); path.pop_back(); }
  void proceed();
  int path_len() { return path.size(); }
  operator bool() { return !path.empty(); }
  vector<C> get_path();
  friend ostream& operator<< (ostream&, const explorer&);
};
```

unordered_set needs to "hash" elements. Relies on a function in class Hash to hash type C objects (can be omitted for standard types for which a default is available).

# Backtracking Code

```
// try to find a tour of length n starting at start
// if fails, returns an empty path
template<class C, class H>
vector<C> find_tour(C start, int n) {
    explorer<C,H> dora(start);
    while(dora && dora.path_len() < n) {
        if(dora.stuck())
            dora.backtrack();
        else
            dora.proceed();
    }
    return dora.get_path();
}
```

classes knight_coords and hasher

```
int main(int argc, char** argv) {
  //... set board size N, starting coords (starti,startj)
  auto start = knight_coords(starti,startj,N);
  auto tour = find_tour<knight_coords,hasher>(start,N*N);
  if(tour.empty())
    std::cerr << "No tour found!" << std::endl;
  else {
    for(auto& c : tour) std::cout << c << " ";
    std::cout << std::endl;
  }
}
```

# The Knight

Demo

- Valid "neighbours" are encoded by the coordinates class

```
class knight_coords {
  char row, col;              // location, as 2 bytes
  const int boardsz;          // board size (alternately, make it static)
public:
  knight_coords(char r, char c, int sz) : row(r), col(c), boardsz(sz);
  vector<knight_coords> reachable();                    // encodes knight's moves
  bool operator==(const knight_coords& other) const; // needed for unordered_set
  friend class hasher;                                  // needed for unordered_set
  friend ostream& operator<< (ostream& out,const knight_coords& kc);
};
```

```
class hasher {
public:
  std::size_t operator() (const knight_coords& kc) const {
   return ( (kc.row << 8) | kc.col ); // a "trivial" hash
  }
};
```

or, `std::hash<int>()((kc.row<<8)|kc.col)`

# Simplecpp Animation

- We will write a separate program for animating a tour
- Reads the tour from its standard input
  - In the same format as output by the tour-finding program
- Can run the two programs together, *piping* the output of find-tour to input of show-tour:

```
$  ./find-tour 7 | ./show-tour 7
```

- Or alternatively, save the output from find-tour in a file, and later animate it

```
$ ./find-tour 7 > 7tour
$ ./show-tour 7 < 7tour
```

# Simplecpp Animation

- Simplecpp has classes for shapes
  - Rectangle, Line, Text, …
  - All derived from a base-class Sprite
- We will have a board which maintains all the squares
- Each square will maintain a piece (possibly empty, or a mark, or a knight)
  - Need to conveniently denote the type of the current piece
  - enum allows defining a type with values which have names

```
enum piece {none, mark, knight};
```

# Simplecpp Animation

```
enum piece {none, mark, knight};
```

```
class square {
    piece P = none;
    double x, y;
    Sprite* img = nullptr;
    void draw(); // change img to hold the shape for current piece
public:
    void init(double x1, double y1, double sqr_side, bool light);
    void setpiece(piece p) {  P = p; draw(); }
};
```

# Simplecpp Animation

Demo

```
class board {
    int n = 8;
    double side, sqr_side, margin=10;
    vector<vector<square>> P;  // all squares with pieces
    int currx=-1, curry=-1;    // active square. uninitialised.
    Line* edge;                // an image separate from squares
    double X(int x) { return margin+(x+0.5)*sqr_side; }
    double Y(int y) { return side+margin-(y+0.5)*sqr_side; }
public:
    board(int n, double side); // init all P[i][j], creates edge
    ~board() { delete edge; }
    void moveto(int x,int y);  // update currx,curry and call setpiece()
};
```