Write and RUN the first C++ program

→ C++ is a compiled language.
→ the compiler converts our code to an executable program that the computer can run.

→ Running a C++ program is normally a two step process.
  ① compile our code with a compiler
  ② Run the executable file that the compiler outputs.

C++ Main()

→ In C++, every program contains a main function which is executed automatically when the program runs.

→ Every part of a C++ program is run directly or indirectly from main, and the most basic program that will compile in C++ is just a main function with nothing else.

→ main() should return an integer, which indicates if the program exited successfully.

Advantages & Disadvantages of compiled languages

→ It catches errors systematically and early
→ It generates much better code.
→ It helps to get the code correct.

Dynamically scripted language

→ Run quickly

C++ output and language Basics

```
#include <iostream>
```

→ preprocessor Directive (Tells the compile to include the library before the compilation

→ It searches for the iostream header file and pastes its content into the program

→ iostream contains the declarations for the input/output stream objects.

iOStream → STD comes with C++

Using std::iout

- Tells the compiler to use the version of cout by the STD
- Namespaces are a way in C++ to group identifiers (name) together. They provide context for identifiers to avoid naming collisions.
- The std namespace is the namespace used for the standard library.
- The using command adds std::cout to the global scope of the program. This way you can use cout in your code instead of having to write std::cout
- cout is an output stream you will use to send output to the notebook or to a terminal, if you are using one.
- << - operator is a stream insertion operator and it writes what's on the right side of the operator to the left side

Strongly Typed languages

⊙ In Strongly typed languages, types are enforced rigidly. once you declare a variable as a certain type, you can't perform operations that don't make sense for that type.

Eg: Java, Python, Rust

Weakly typed language

Here, types are more flexible, allowing implicit type conversions. This flexibility can make the code easier to write quickly but may lead to unexpected behaviours as the language may convert between types without explicit instruction.

Eg: Javascript

Primitive Variable Types

- int
- float
- String

```
#include <iostream>
#include <string>
using std::cout;

int main()
{
  int a = 9;
 std::string b;
}
```

Vector

→ a linear sequence of contiguously allocated memory
→ It allows dynamic sizing unlike arrays.
→ Flexible capacity

Vector Containers

ID Vector :- C++ has several container types that can be used for storing data.

∴ Vectors are a sequence of elements of a single type, and have useful methods for getting the size, testing if the vector is empty, and adding elements to the vector.

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

int main()
    {
#three ways of initialising and declaring vectors.
std::vector<int> v_1 {0,1,2};
```

```
std::vector<int> v_2 = {3,4,5};
std::vector<int> v_3;
  v_3 = {6};
}
```

2D vector

:) unfortunately there is not a built in way to print vectors in c++ using cout.

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

int main()
  {
//creating a 2D vector

std::vector<std::vector<int>> v;
  v = {{1,2}, {3,4}};
```

Comments

// You can use two forward slashes for single line comments

/* You can use this for multi-line comments.
. . . . . .
*/

auto

- Usually the type of each variable was explicitly declared.
- In genral, that is not necessary, and the compiler can determine the type based on the value being assigned.

Accessing Vectors

1D vector

```
std::vector<int> v_1 {0,1,2,3};

cout << v_1[0];
cout << v_1[1];
cout << v_1[2];
```

2D vector

→ In vector, if you try to access the elements of

a vector using an out-of-bound index, you might have noticed that there is no error or exception thrown.

→ In this case, the behaviour is undefined, so you cannot depend on a certain value to be returned.

length of vector

- size()

```
std::vector<int> a;
a = {1,2,3,4};
std::cout << a.size();
```

↳ prints 5

for 2D vector

```
cout << b[0].size();
```

```
cout << b[i].size();
```

loops

A simple for loop using an index variable has the following syntax.

```
#include <iostream>
int main()
    {
    for (int i=0; i<5; i++) {
        cout << i << "\n";
        }
    }
```

Iterate over through a container

C++ offers several ways to iterate over containers. One way is to use an

→ index-based loop

→ Range based loop

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

int main()
    {
std::vector<int> a={1,2,3,4,5};
for (int i : a) {
        cout << i << "\n";
        }
    }
```

Challenge

Range based loop for a 2D vector.

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

int main()
    {
for (auto v : b) {
      for (int i : v)
            {
        std::cout << i << " ";
            }
      cout << "\n";
      }.
```

Functions

The basic syntax of a function is

```
return-type Functionname (parameter_list) {
    ... }.
```

```
#include <iostream>

int addTwoNumbers (int i, int j)
    {
     result i+j ;
    }
int main()
    {
     int i=3:
     int j=4:
   std::cout << addTwoNumbers (3,4) ;
    }.
```

Void Return Type

→ Sometimes a function doesn't return anything.

→ If a function doesn't return a value, the void type can be used for the return type.

```
Void foo(int a, int b) {

    std::cout << a+b;
}
```

If Statement and while loops

→ In c++, the boolean conditions is contained in parenthesis (.) and the body of the statement is enclosed in curly brackets. { }

```
#include <iostream>

int main()
  {
   bool a = false;

if (a) {
 std::cout << "Hello world";
 }}
```

While loop

```
while (condition)
    {
    ...
    }
```

```
#include <iostream>
int main()
{
int i=0;
while (i<5)
{
cout << i << "\n";
i++;
}}
```

Reading from a file

Creating an input stream Object

→ In C++, you can use the `std::ifstream` object to handle input file streams.

→ To do this, we need to include the header file that provides the file Streaming classes.
`<fstream>`

```
std::ifstream my_file;
my_file.open(path);
```

→ C++ ifstream Objects can also be used as a boolean to check if the stream has been

created successfully.

→ If the stream were to initialize successfully, then the ifstream object would evaluate to true. If there were to be an error opening the file or some other error creating the stream, then the ifstream object would evaluate to false.

```
#include<iostream>
#include<fstream>
#include<string>

int main()
  {
  std::ifstream my_file;
  myfile.open("files/1.baud");
  if (my_file)
      {
  std::cout<<"file opened"<<"\n";
  }
  }
```

Reading Data from the Stream

→ If the input file stream object has been sucessfully created, the lines of the input stream can be read using the getline method.

```
#include <iostream>
#include <fstream>
#include <string>

int main()
{
  std::ifstream my_file;
  my_file.open("pas.txt");
  if (my_file)
  {
    std::cout << "File Stream created" << "\n";
    std::string line;
    while (getline(my_file, line))
    {
      std::cout << line << "\n";
    }
  }
}
```

Processing Strings

→ In order to process the string, rather than streaming it to cout, there are many options available.

→ Eg: istringstream header from <sstream>

Streaming int's from a string with istringstream

→ In c++, strings can be streamed into temporary variables, similarly to how files can be streamed into strings.

→ Streaming a string allows us to work with each character individually.

→ One way to stream a string is to use an input string stream object istringstream from the <sstream> header.

→ Once an istringstream object has been created, parts of the string can be streamed and stored using the "extraction operator": >>

→ The extraction operator will read until white-spaces is reached or until the stream fails.

```
#include <iostream>
#include <sstream>
#include <string>

int main() {

std::string a("1 2 3");

isstringstream my_stream(a);
 int n;
  my_stream >> n;
  cout << n << "\n";
```

The extraction operator >> writes the stream to the variable on the right of the operator and returns the isstringstream object, so the entire expression `my_stream >> n` is an isstringstream object and can be used as a boolean. Because of this, a common way to use isstringstream is to use the entire extraction expression in a while loop.

```
#include <iostream>
#include <string>
#include <sstream>

int main() {
  String a("1 2 3");

  istringstream my_stream(a);
  int n;

  while (my_stream >> n)
        {
  std::cout << "Success" << "\n";
        }
  Std::cout << "Failure";
      }
```

Vector push_back()

→ The result of the above example can be stored in a container of our choice (vector).

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

```
int main()
    {
    std::vector<int> v {1,2,3};

    for(int i=0; i<v.size(); i++)
            {
      std::cout << v[i];
            }
    v.push_back(4);

    }}
```

→ String streaming objects are very powerful in C++, and there are more ways than an istringstream.

Enumerator (enum)

- Enum is a way to define a type in C++ with values that are restricted to a fixed range.

```
#include <iostream>

int main()
    {
    enum class color {          //scoped enums
        white,
        blue,
        black,
        red
          };

    Color my_color = color::red;

    If (my_color == color::red)
        {
      std::cout << "car is red" << "\n";
        }
     else {
       Std::cout << "car is not red";
         }}.
```

Example with a Switch Statement

```
#include <iostream>
```

```
int main()
    {
enum class Direction {
        kUP,
        kdown,
        kleft,
        kright.
         };

Direction a = Direction :: kUP.

Switch (a)
    {
 case Direction :: kUP:
      Std::cout << " Going up";
      break;
 case Direction :: kdown:
      Std::cout << "Going down";
       break;
 case Direction :: kleft :
      Std::cout << "Going left";
       break;
 case Direction :: kright :
      Std::cout << "Going right";
       break;
    }}
```

Re-cap

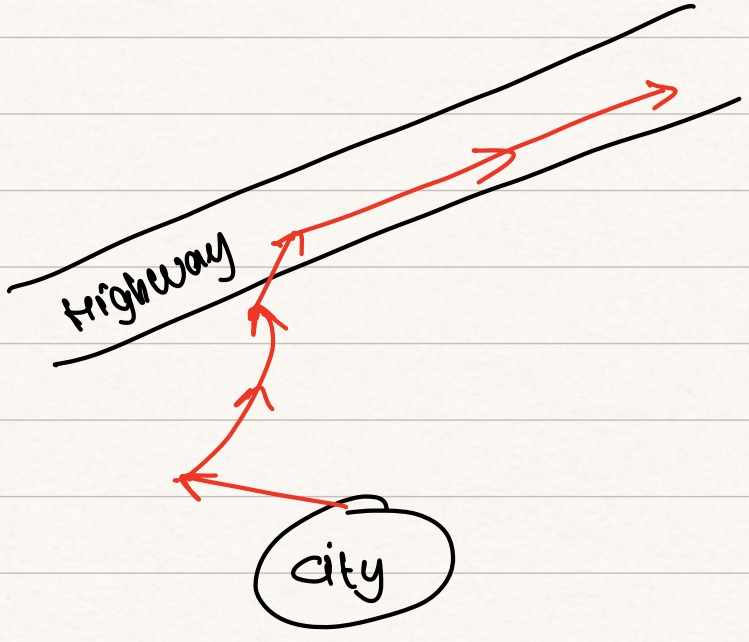
- How to compile and run a simple c++ program
- Send output to the terminal
- Variables and containers
  - Variable types
  - Vectors
  - auto
- Functions and control structures
  - Conditions
  - LOOPS
  - Functions
- Data input
  - Read data from a file
  - Parse data and process strings
- Define your own types with enums.

A* Search (Algorithm)

→ A* Algorithm is an algorithm that is frequently used for path finding when working with graphs.

→ A* search allows us to efficiently find a path, if one exists, between any two nodes in the graph.

Motion planning



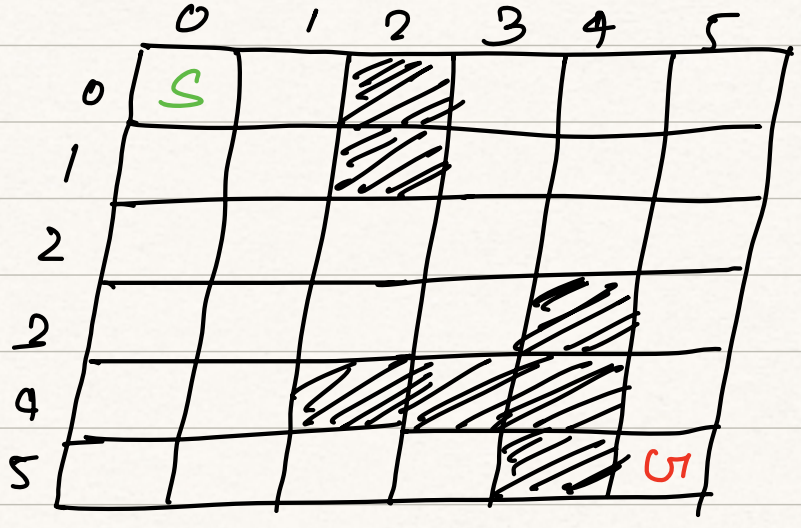
→ The process of find a path between start location and goal location is called planning.

Planning Problem

Given : Map
Starting location
Goal location
Cost function

Goal :- find minimum cost path

path-planning - Search



S - start
G - Goal

→ Problem: Find shortest path between S and G?

*) Split the area into grid and name the cells.
*) Keep the list of cells which seems interesting to us to search.

Eg: open = [0,0]1 (first cell)

*) Check whether the cell is my goal. If not expand the list.

*) open = [1,0]1 [0,1]1

*) red indicates how many expansions it took to reach that cell. (g-value)

=

*) The g-value is with the red, which is the expansion. proceed with one having small g-value. Above both have same g-value.

*) Consider [1,0], it has three neighbours, [0,0] is already done. So [2,0] and [1,1]

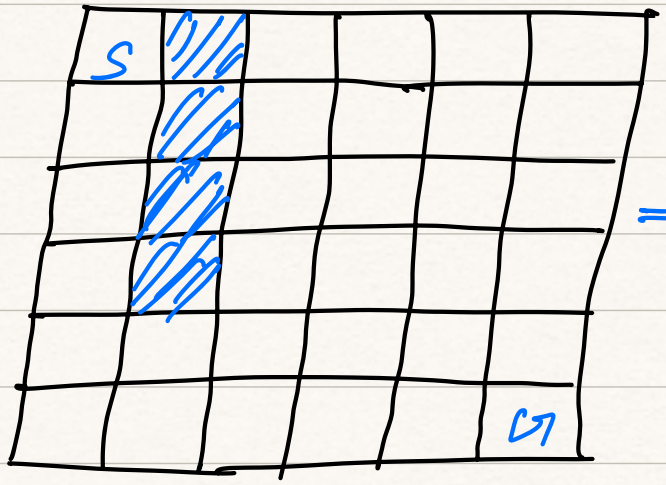
open = [2,0]2 [1,1]2

*) It will continue and reach the goal position. The g-value will give the steps it took from start to finish.

→ In the above case, it is a general path search which works between two cells on a board.

→ There is an improvised way of searching using an algorithm called **A* search**

A* Search



⇒

```
[[0, 1, 0, 0, 0, 0]
 0, 1, 0, 0, 0, 0,
 0, 1, 0, 0, 0, 0,
 0, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0]
```

0 → navigable space.
1 → occupied space.

→ A* uses a Heuristic function $h()$

i.e. $h(x,y) \leq$ distance to goal from cell (x,y)

→ In this approach we start as earlier.

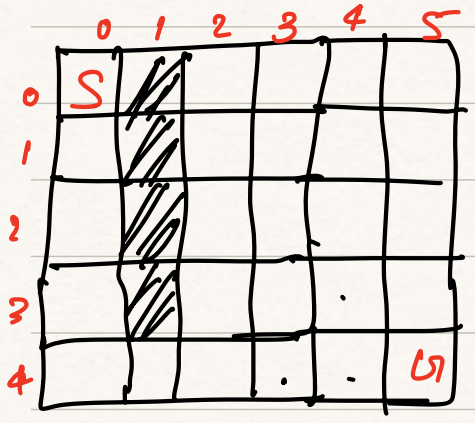
open = [0,0] $0+h(x,y)$ ⟹ [0,0]$_q$

Since heuristic value of (0,0) is 9. There is a table available with $h(x,y)$ for each cell.

$$f = g + h(x,y)$$

→ Remove the element with lowest 'f' value instead of 'g'.

Eg: consider if now at open = [4,1] $5+4=9$ ($g+h(x,y)=f$)



open = [4,2] $6+3=9$

[3,2] $7+4=11$ [4,3] $7+3=10$

[3,3] $8,3$ [4,4] $8,1$

[3,4] [4,5] $9,0$ Goal state

→ By providing additional information i.e. Heuristic function the goal can be achieved easier.

Summary and A* Search Pseudocode

→ A* Search algorithm is similar to breadth-first search, except for the additional step of computing a heuristic and using that heuristic (in addition to the cost) to find the next node.

Pseudocode

Search(*grid*, *initial_point*, *goal_point*) :

1. Initialize an empty list of open nodes.
2. Initialize a starting node with the following:
   x and y values given by initial_point*.
   ***g_value** = 0, where g is the cost for each move.
   ***h_value** given by the heuristic function (a function of the current coordinates and the goal).
3. Add the new node to the list of open nodes.
4. **while** the list of open nodes is nonempty:
   a. Sort the open list by **f_value**, which is equal to the summation of the **g_value** and the **h_value**.
   b. Pop the optimal cell (called the *current* cell).
   c. Mark the cell's coordinates in the grid as part of the path.
   d. **if** the *current* cell is the goal cell:
   return the grid*.
   e. **else**, expand the search to the *current* node's neighbors. This includes the following steps:
   Check each neighbor cell in the grid* to ensure that the cell is empty: it hasn't been closed and is not an obstacle.
   *If the cell is empty, compute the cost (**g_value**) and the heuristic, and add to the list of open nodes.
      - Mark the cell as closed.
5. If you exit the while loop because the list of open nodes is empty, you have run out of new nodes to explore and haven't found a path.

Search(grid, initial_point, goal_point):

1) Initialize an empty list of open nodes.
2) Initialize a Starting node with the following:
   - x and y values given by initial_point.
   - g_value = 0, where g is the cost for each move
   - h-value given by the heuristic function
3) Add the new node to the list of open nodes
4) While the list of open nodes is not empty.
   - Sort the open_list by f-value which is equal to the summation of the g_value and the h-value.
   - pop the optimal cell (called current cell)
   - Mark the cell's coordinate in the grid as part of the path
   - If current cell is the goal cell, then return the grid.
   - else
     expand the search to the current node's neighbours.
     This includes the following:-
     → check each neighbour cell in the grid to ensure that the cell is

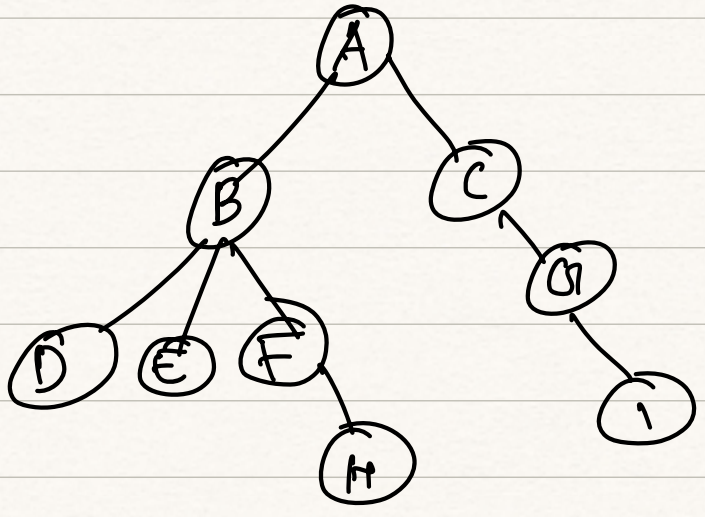
empty. It hasn't been closed and is not an obstacle.

→ If cell is empty, compute the cost (g-value) and the heuristic and add to the list of open nodes.
:) Mark the cell as closed.

5) If you exit the while loop because the list of open nodes is empty, you have run out of new nodes to explore and haven't found a path.

Extras

→ Breadth first Search (BFS) :- Algorithm for searching a graph.
→ Breadth = broad / wide (so horizontal first)
→ Use of queue data structure (FIFO)



Start from A, it goes

Queue: [A]

↑ pop & mark as visited

: []

(FIFO) : [C, B]

↑ pop

: [F, E, D, C]

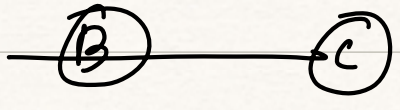
↑ pop

[F, E, D]

↓

[G, F, E, D]

→ Here we proceed/visited all the nodes in the horizontal before moving vertically down

ie 

[G, F, E] pop

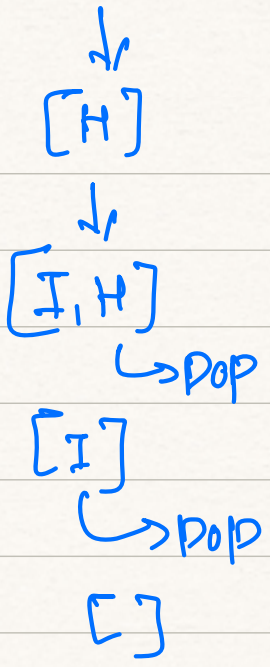
↓

[G, F] → pop

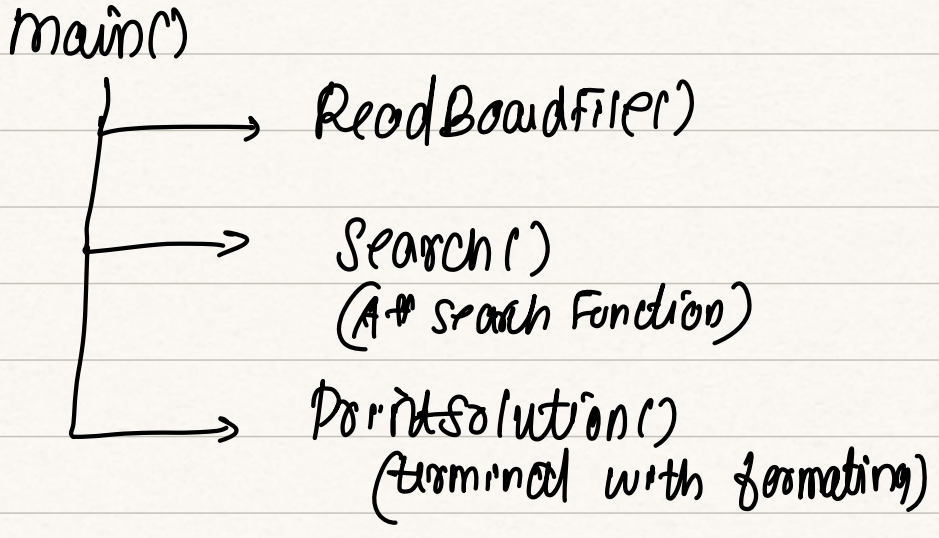
[G]

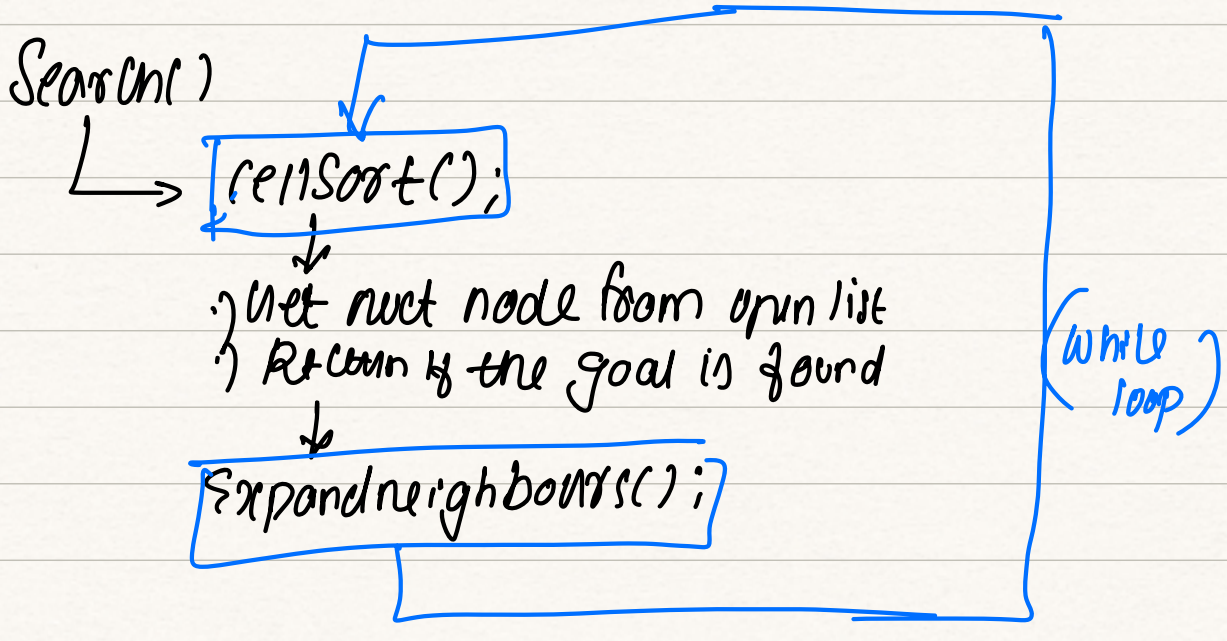
↓

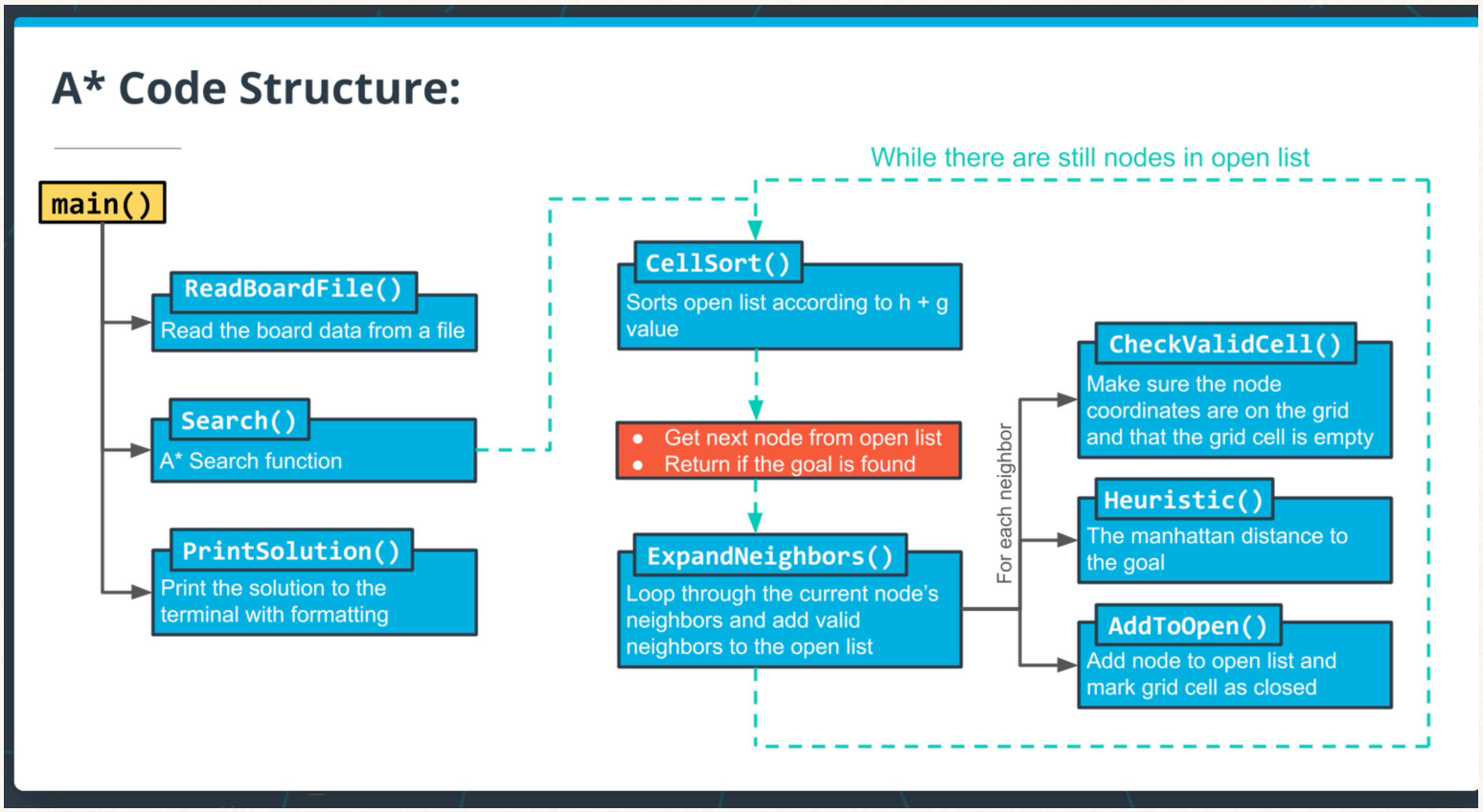
[H, G] → pop



A* code Structure







Summary

*) cellSort() – Sorts the open list according to the sum of h+g.

*) ExpandNeighbors() – loops through the current node's neighbours and calls appropriate functions to add neighbours to the open list

*) CheckValidCell() – ensures that the potential neighbour coordinates are on the grid and that cell is open.

*) Heuristic() – computes the distance to the goal.

*) AddToOpen() – adds the node to the open list and marks the grid cell as closed.

Manhattan ometry (Taxicab ometry)

In $\mathbb{R}^2$, the manhattan/taxicab distance between $P = (P_1, P_2)$ and $Q = (Q_1, Q_2)$ is

$$|P_1 - Q_1| + |P_2 - Q_2|$$

Pass by Reference

→ When a function is called on some data, a copy of that data is made, and the function operates on a copy of the data instead of the original data. This is refered to as pass by value, since only a copy of the values of an object are passed to the function, and not the actual object itself.

Constants

→ C++ supports two notions of immutability :-

Const :- meaning roughly "I promise not to change this value"!... The compiler enforces the promise made by const ...

Constexpr :- Meaning roughly "to be evaluated at compile time". This is used primarily to specify constants.

→ constexpr guarantees that the value is evaluated at compile time, making it ideal for cases where you absolutely need the result at compile time such as array sizes, template arguments, or switch cases.

→ const doesnot guarantee compile-time evaluation.

```
#include <iostream>

int main()
    {
    int i;
    std::cout << "Enter an integer";
    std::cin >> i;

    const int j = i * 2;   // j can only be evaluated at run time.
                           // But it promised not to change it after it is initialized.
```

```
constexpr int k = 3;        // k can be evaluated at compile time.

std::cout << "j:" << j;
std::cout << "k:" << k;
```

compiler catches if we try to change the value.

```
int main()
  {
  const int i = 2;
  iff:                      // illegal
  }

int main()
  {
  constexpr int i = 3;      // illegal.
  iff;
  }
```

- A common use of const is to guard against accidently changing a variable, especially when it is passed-by-reference as a function argument.

Writing Multiple files

- Header Files
  - → using header files to break a single file into multiple files.

- Build systems
  - → CMake and Make

- Tools for writing larger programs
  - → References
  - → Pointers
  - → Maps
  - → Classes and object oriented programming

Header files

Header Files or .h files, allow related function, method, and class declarations to be collected in one piece. The corresponding definitions can then placed in .cpp files. The compiler considers a header declaration a "promise" that the definition will be found later in the code, so if the compiler reaches a function that hasn't been defined yet, it can continue on

Compiling until the definition is found. This allows functions to be declared in arbitrary order.

→ Header files also solve the problem of ordering of the function. One way to solve the problem is to declare the function on top of the program.

→ To avoid a single file from becoming cluttered with declarations and definitions for every function, it is customary to declare the functions in another file, called the header file.

Eg:
```
#ifndef HEADER_EXAMPLE_H
#define HEADER_EXAMPLE_H

Void OuterFunction (int);
Void InnerFunction (int);

#endif
```

→

Preprocessor directive

```
#ifdef HEADER_EXAMPLE_H
#define HEADER_EXAMPLE_H
```

} include guard

→ The include guard prevents the same file from being pasted multiple times into another files.

→ #pragma once is another preprocessor directive

CMake and Make

→ As said earlier, the programs can be split into multiple .h and .cpp files, but when build we have to include all together.

For eg: `g++ -std=c++17 main.cpp Vectorsrm.cp && ./a.out`

→ For small projects, it works but for larger projects, it will be difficult to type all the

Name of the files in the command line.

→ Many larger c++ projects use a build system to manage all the files during the build process.

→ The build system allows for large projects to be compiled with a few commands, and build systems are able to do this in an efficient way by only recompiling files that have been changed.

Learn about:

→ what actually happens when you run g++

→ How to use Object files to compile only a single file at a time. If you have many files in a project, this will allow you can compile only files that have changed

Object file

When we compile a project with g++, it actually performs several distinct tasks:

1) preprocessor runs and execute any statement beginning with a hash symbol #.

This ensures all codes is in the correct location and ready to compile.

2) Each file in the source code is compiled into an "object file" (a.o file). object files are platform specific machine code that will be used to create an executable.

3) The object files are linked together to make a single executable. In the examples you have seen so far, the executable is a.out

In order to do in separate steps:-

1) g++ -c main.cpp

↓

main.o (produces)

2) g++ main.o

↓

produces an executable.

(a.out → unix based

a.exe → windows based)

main.cpp

```
→ g++ -c main.cpp
→ g++ main.o -o main.out
→ ./main.out
```

→ compile a main.cpp with custom output name.

Mutiple file compilation

- multiple_files_example
  - Increment_and_sum.cpp
  - Increment_and_sum.h
  - main.cpp
  - Vector_add_one.cpp
  - Vector_add_one.h

to compile:

```
g++ -c *.cpp
g++ *.o
./a.out
```

* will take all files with the corresponding extension.

→ For larger projects, always good to use a build system, which can compile exactly the right files for you and take care of linking.

CMake

→ Open source, platform independent build system.
→ Use CMakeLists.txt

→ In general, cmake only needs to be run once for a project, unless you are changing build options.
(eg: Using different build flags or changing where you store your files).

→ Once finish CMakeLists.txt

```
mkdir build
cd build
cmake ..
make
```

→ produces executable
./example

Toy: change main.cpp and save it.
run make

Reference

→ A Reference is another name given to an existing variable. '&' operator can be used to declare a reference.

Pointers

→ A pointer is a variable that stores the memory address of an object in your program.

→ They keep track of where the variable is stored in the computer's memory.

→ The memory address can be accessed using an '&' symbol

```
#include<iostream>

int main()
  {
  int a;
  int b;
  std::cout << &a; //gives address of 'a'

  }
```

→ At this point, you might be wondering why the same symbol & can be used to both access memory addresses and pass by references.

The overloading of & and * probably contribute to much of the confusion around pointers.

The symbol & and * have different meaning, depending on which side of an equation they appear.

→ [For the & Symbol if it appears on the left side of an equation, it means that the variable is declared as a reference.]

→ If the & appears on the right side of the equation or before a previously defined variable it is used to return a memory address.

> leftside : Variable declared as a reference.
> Rightside : Return a memory address.

```
int i=5
int* ptr = &i;  // Returned the address of .i.
```

Pointers to objects

```
class X
  {
public:
 int value;

 void display()
     {
    std::cout << value;

    };

int main()
   {
 X* d1 = new X;
 d1-> value = 10;
 d1-> disp();
  }
```

// dynamic memory allocation.

// d1 is a pointer to an Object of class X.

Pointers to other Object Types

Vector

```
#include <iostream>

int main()
{
    std::vector<int> value {1,2,3};

    std::vector<int>* ptr = &value;

    for(auto& i : *ptr)
    {
        std::cout << i;
    }
```

```
std::cout << (*ptr)[0];

std::cout << ptr->at(0);
```

here

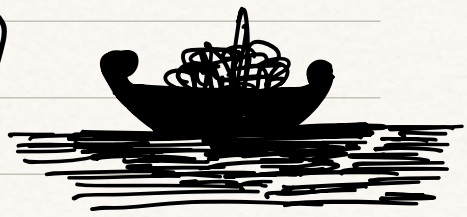
ptr is a pointer to vector value. So in order to access member function we must use →

```
ie      ptr->at(0);
```

But in order to use dot operator (.)

We must dereference it first and use it

ie `(*ptr).at(0);`



`ie (*ptr)[0];` ← accessing member.

PASSING POINTERS FROM A FUNCTION

```
int * Addone( int &j)
      {
       j++;
    return &j;
      }

int main()
    {
   int i = 10;
  int* ptr = Addone(i);
  std::cout << *ptr;
```

References v/s Pointers

→ Pointers and References have similar use cases in C++.

References	Pointers
*) References must be initialized when they are declared. This means that a reference will always point to data that was intentionally assigned to it.	*) Pointers can be declared without being initialized, which is dangerous. If this happens mistakenly the pointer could be pointing to an arbitrary address in memory and data associated with that address could be meaningless.
*) References can't be null. This means that a reference should point to meaningful data in the program.	*) Pointers can be null. eg: nullptr

* When used in a function for pass by reference, the reference can be used just as a variable of the same type would be.

* When used in a function for pass-by-reference, a pointer must be dereferenced in order to access the underlying object.

→ References are generally easier and safer than pointers. As a decent thumb of rule, references should be used in place of pointers where ever possible.

Maps

→ container data structures are fantastic for storing ordered data, and classes are useful for grouping related data and functions together, but neither of these data structures is optimal for storing associated data.

* A Map (hashtable, hash map or dictionary) is a data structure that uses key|value pairs to store data, and provides efficient lookup and insertion of the data. The name "dictionary"

Should provide an excellent idea of how these works, since a dictionary is a Real life example of a map.

Eg:

Apple :- It is a fruit normally sweet.

Apple → Key

It is a fruit normally sweet. → Value.

Key: String

Value: Vector<string>

→ std::unordered_map is the c++ standard library implementation of a map.

Syntax :-

```
Unordered_map <key_type, value_type>
    variable name ;
```

ie `std unordeul_map <sting, int> v;`

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

```
#include < string>

int main()
    {
std::string key = "word";
std::string def1 = "a unit of language";
std::string def2 = "a speech or talk";
std::string def3 = "a short talk or conversation";
 std::string def4 = " an expression or utterance";

std::unordered_map <std::string,std::vector<std::
                        string>> my_dict;

if( my_dict.find (key) == my_dict .end ())
    {
std::cout <<" The key is not found";
std::cout << "insert a key-value pair";

my_dict[key] = std::vector<std::string>
                    { def1, def2, def3, def4};
    }

auto definition = my_dict[key];
```

```
for (String definition : definitions)
        {
        std::cout << definition;

        }
```

Object oriented programming

→ It is a style of coding that collects related data (object attributes) and functions (object methods) together to form a single data structure, called object.

→ This allows that collection of attributes and methods to be used repeatedly in our program without code repetition.

code with objects

```
#include <iostream>
#include <string>

class car
    {
    Public :
    void PrintCarData()
```

//method to print data

```
    }
    cout << "The distance that the " << color << " car "
    << number << " has travelled is " << distance;
    }

    void incrementDistance()
        {
        distance++;
        }

std::string color;
    int distance;
    int number;
    };

    int main()
        {
    Car car_1, car_2, car_3;

    car_1.color = "green";
    car_2.color = "red";
    car_3.color = "blue";

    car_1.number = 1;
    car_2.number = 2;
```

```
car_3.number = 3;

car_1.IncrementDistance();

car_1.PrintCarData();
car_2.PrintCarData();
car_3.PrintCarData();
}
```

→ Here the main() function doesn't seem to be organised. This can be improvised by adding a constructor.

* The constructor allows you to instantiate the new objects with the data that we want.

```
#include <iostream>
#include <string>

class car
    {
  public:
    void PrintCarData()
        {
    std::cout << color << number << distance;
```

```
}
void incrementDistance()
{
distance++;
}
```

```
Car(string c, int n)
{

color = c;
distance = n;
}
```

constructor

```
std::string color;
int distance;
int number;
};

int main()
{

Car car_1 = Car("green", 1); or Car car_1("green", 1);
Car car_2 = Car("red", 2);
Car car_3 = Car("black", 3);

}
```

Note :-

The above constructor can also write as

```
car (string c, int n : color(c), distance(n))
    {
    }
```

→ So using an object of class we can access data members using '.' operator.

i.e Eg: myObj.PrintData()

→ Also using a pointer of the object, we can do the same using :

myPtr → PrintData()
or
(*myPtr).PrintData()

* → dereference operator
→ arrow operator

```
int main()
{
car myCar;
car* myPtr = &myCar;
```

```
myObj -> printData();
}
```

Inheritance

→ It is possible for a class to use methods and attributes from another class, using class inheritance.

```
class Sedan : public Car
{
- - -
- - -
- - -
};
```

→ Here each Sedan class instance (object) will have access to any of the public methods and attributes of the car.

Car → parent class.
Sedan → child or derived class.

Note :-

→ (1) When the class methods are defined outside the class, the scope resolution operator :: must be used to indicate which class the method belongs to

Eg: Void car::PrintCarData();

This prevents any compiled issues if there are two classes with same method name.

(2) Constructor can use a initialization list as mentioned above

```
car(String c, int d : color(c), distance(d))
{
}
```

(3) Variables that don't need to be visible outside of the class are set as Private. This means that they can not be accessed outside of the class which prevents them from being accidentally changed.

So in order to separate the files.

car.h

```
#ifndef CAR_H
#define CAR_H

#include <iostream>
#include <string>
```

car.h

```
class car
    {
public:
    void PrintCarData();
    void IncrementDistance();

    car(string c, int d : color(c), number(d)
        { }

private:
    std::string name;
    int distance = 0;
    int number;
    };

#endif
```

car.cpp

```
#include <iostream>
#include "car.h"

class car
    {
```

```
void car :: PrintCarData ()
        {
    std::cout << color << number;
        }

void car :: IncrementDistance()
        {
    distance ++;
        }
```

```
#include <iostream>
#include <string>
#include "car.h"

int main()
    {
car car_1 = car("green", 1);
car car_2 = car("red", 2);

car_1. IncrementDistance();

car_1. PrintCarData();
}
```

For 100 cars, the object oriented car_main.cpp will be extremely useful.

```
#include <iostream>
#include <string>
#include <vector>
#include "car.h"

int main()
    {

std::vector<car*> car_vect;
 Car* cp = nullptr;

std::vector<std::string> colors{"red",
                                "green",
                                "Blue"};

//create 100 cars with different colors and
  push pointers to each of those car into
  the vector.

for(int i=0; i<100; i++)
    {
```

```
cp = new Car(colors[i%3], i+1);
car_vect.push_back(cp);
    }

for(Car* cp : car_vect)
    {
  cp -> incrementDistance();
    }

  for(Car* cp : car_vect)
      {
   cp -> printCarData();
      }
  }
```

This Pointer

→ 'this' pointer points to the current class instance or object.

ii) we can use this-> to access the member functions & member variables.

```
Void PrintData()
    {
Std::cout << this->color;
    }
```

Build an OpenStreetMap Route planner

→ To create a route planner that plots a path between two points on a map using real map data from the OpenStreetMap project.

→ Write the project in C++ using real map data and A* search to find a path between two points.

→ In this project, an OSM XML file (.osm) is used

Data Types overview

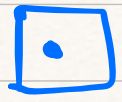
→ In the XML element types, there are 3 element types, which are important to the project.
- → Node
- → Ways
- → Relations.

Node

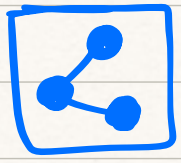
→ A Node is one of the most basic elements in the OpenStreetMap data model. Each node includes

a single point with an identifier (id), latitude (lat), and longitude (lon). There are other XML attributes in a node element that won't relevant to the project such as (User) and the (timestamp) when the node was added to the data set.

→ It is used to mark locations. Nodes can be separate or can be connected.



Ways

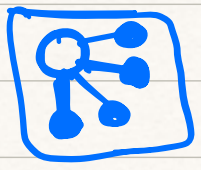


- A way is an ordered list of nodes that represents a feature in a map. This feature could be a road, or a boundary of a park, or some other feature in the map.

- Each way has at least one (tag) which denotes some information about the way, and each way also belongs to at least one relation.

→ open ways
→ closed ways
→ Areas.

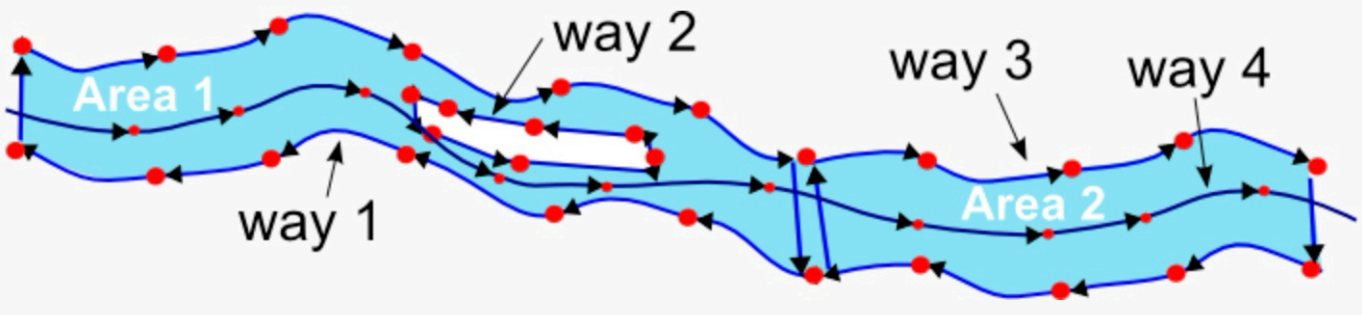
Relation



1) A relation is a data structure which documents a relationship between other data elements.

2) A route relation which lists the ways that form a major highway, cycle route, or bus route.

Node
Way
Relation



In the above figure, there are

→ 'n'-nodes (Red points)
→ 4-ways (Close, open, avail)
→ Rectangle