



CS 302

**Assignment 12: Final Project**  
Counter - Strike: Valorant Offensive  
Dijkstra's Shortest Path for Pathfinding

*Dread it. Run from it. Destiny arrives all the same. And now, it's here. Or should I say, your Final project is here.*

Your **assignment** is easy. Code a fully functional 2D version of Competitive CS:GO mode minus the Weapon Economy (just one gun everyone spawns with) and the Online multiplayer (1 human, 9 bots, or all bots). By 2D I mean the game is to be played on a top down view as if looking at it from birds view. The game must run on the terminal and must be coded from scratch. You may use the ncurses library to make user input work better.

That's it!

*gg ez*



For those of you not familiar with Counter-Strike: Global Offensive (CS:GO), Valorant, Call of Duty: Search and Destroy, or the latest flavor of this popular online competitive E-sport First Person Shooter (FPS) gameplay, let me familiarize you with the basics:

- There are two teams. Attackers and Defenders. In CS:GO the attackers is usually the Terrorist(T) team and the Defenders are usually the Counter Terrorist(CT) team. Each team has 5 players.
- The job of the attackers (T) is to plant a bomb on one of the bombsites that the CT have to defend.
- Depending on the level, also known as map, there may be one, two or sometimes three bombsites (Valorant) sites to defend. A bombsite has a specific marked area where a T player can plant the bomb.
- There is only one bomb and it can only be carried by T players. If the player carrying the bomb is killed he drops in his last location and the remaining T players have to pick it up to try and plant.
- If the T players successfully plant the bomb, a 30 second fuse is lit in which the CT players must go and defuse the bomb. On the other hand the T players will defend the bombsite to prevent CT from defusing!
- The round ends when one of five things happen:
  - The T team plants the bomb and the bomb explodes after the 30 second fuse. Regardless of who is left alive at that point
  - At any point if the T team kills the entire CT team, they win the round whether bomb has been planted, or not.
  - The CT team wins if the entire T team is killed and bomb has not been planted.
  - The CT team wins if the entire T team is killed and bomb is defused if it was already planted before the last T player was killed. In other words it's possible that if the last T player plants the bomb and then defends it for maybe 25 seconds and is then killed but the CT team does not get to defuse the bomb before the timer runs out.
  - On the other hand if a T player plants bomb and runs away, the CT team could defuse the bomb and even though there are remaining T players alive, CT automatically wins the game after defusal.
  - Finally the rounds last 1 minute 55 seconds and if the timer runs out the defenders (CT) win the round.
  - Planting takes about 5 seconds and Defusing takes 10 seconds (5 with a defuse kit), but in our game there are no kits and you can make plant / defuse instant or delayed. Up to you.
- Just to emphasize, When a player has been killed he does not respawn until the following round.
- Because the CT have to defend, on real competitive play they usually play defensive in the sense they just hold back and wait for enemy to attack, having cover to help them. On the other hand T have the element of surprise in that they can rush one site as a full team and overwhelm CT since they have to divide their 5 players into multiple bomb sites.
- Maps tend to be balanced but it does not matter since on a regular competitive match each team plays 15 rounds as attackers/defenders and then they swap sides. First to 16 rounds wins the match and continues

their road to Global Elite! (The ranking system). In your implementation you only need to make a single round game.

- I shouldn't go on without saying that each team has weapons and they can kill the enemy by shooting them. However, friendly fire is also on which means you can kill those useless teammates too, lol. When a player shoots a bullet the game must track its location until it hits an object or a player that can take damage.

So that's the basics. To emphasize here are the differences from real CS:GO and our CS:VO:

- While CS:GO is played 5v5, usually it is 5 humans vs 5 humans (via online or LAN multiplayer) unless a player disconnects. In which case a bot (artificial intelligence) takes over and plays the round. If a human player dies, they can take over the bot if it is still alive. In our implementation we only allow 1 human, or 0 humans and as a result 9 or 10 bots (9 bots and 1 human or all 10 players are bots). And if the human dies, he cannot take over and control any of the bots.
- The entire program is a single round of CS:GO rather than the usual 16-30 round matches.
- At the start of the game you pick if you want to play or just let the bots play on their own. If you do play you get to pick your team. (In real CS:GO matchmaking puts you together but also you can team up with friends)
- CS:GO has a wonderful economy system for buying a varied and balanced  array of weapons and utility that involves rewards for kills or bomb plant/defusals, and round win/loss bonuses. To make things more streamlined we are removing all of that and having each player (and by player I mean human or bot) spawn with a single gun that shoots in Semiautomatic mode. (each depression of the trigger/key results in one bullet fired) But feel free to make your version have more weapon options or Full-auto mode for guns.
- CS:GO is an FPS meaning you play as if you were looking through the eyes of the character. In this case we are making it a top down 2d version of the game similar to if a bird was looking at the entire map from the sky.
- CS:GO has amazing graphics , we are instead going to stick to using ASCII and the terminal to render our entire game. The input file for the game will be a text file of a specific set of ASCII characters that provide the map layout. We'll talk about this file in more detail later on.
- As far as my understanding goes, the AI in CS:GO works via pre-define paths that it may choose to take depending on its objective goal. In our case we are generating all pathfinding on the fly and regenerating for each step we take depending on our goal. This means that the Bots should be able to handle any map file given in the command like argument. So our bots can truly adapt 😊
- I know some of you have little experience with the ncurses library so while CS:GO normally works with asynchronous input: meaning that the AI and the entire game runs whether or not you move, in our version the game pauses each tick and waits for our input then it runs 1 tick by doing the input we gave it plus also having the entire game's entities (bots or projectiles) update by that 1 tick. A great example that someone gave me on Discord is the game Superhot VR. It works the same way with the environment only moving when you do.

Next let's talk about the input file (image next page)

First, the name of the file is what you will feed to your program when you run it. For example, for the map I did it would be,

```
./a.out sjf_dust2.txt
```

The file is made up of a specific subset of characters that we can then parse when we launch the game to build the environment. For example. The character x is used to signify walls that the players or bullets cannot walk over. Bullets will despawn if they hit them as well. On the other hand, the spaces signify walkable space and so on:



Key:  
#: Can only go horizontal or vertical depending on dir entered (ghetto mimic of tunnel/bridge)  
x: Wall (cannot shoot, see, or go through)

First note that the first line contains a single set of character (no spaces) to signify the map name. In this case DUST\_2. Next is the chracter height, followed by an x followed by the character width of the map. Technically the dimensions could be computed on the run but I thought to make life easy by having it be there.

Then comes the map after which any text after the map is ignored and can be used for comments or documentation.

The full key of characters available to use when creating a map is as follows:

- **#**: Can only go horizontal or vertical depending on the direction player was when they entered into this type of character. The reason for this is because we can mimic a bridge/tunnel with it. AI cannot see in directions perpendicular to its path.
- **x**: Wall (cannot shoot, see, or go through and bullets despawn when you hit a wall)
- **o**: Obstacle (AI can see behind it and shoot bullets that will go through it, but no player can walk through it. Note that this is a lower case o as in Oscar.
- **BLANK SPACE**: The blank spaces you see – and make sure they are spaces and not tabs if you are making your own maps – is open space which means players can see, shoot, and walk through in any vertical, or horizontal direction.
- **B**: This is where the bomb spawns. Once the game is running if the bomb is carried by a player then it disappears, but if a player dies and drops it or if they plant it then it will show up with a B on the map to show the location.
- **C**: Counter Terrorist (Defenders) Spawn Location. It also signifies an active C bot (AI) player. Because more than one player can be in the same coordinate. It will only show 1 letter at the start when all 5 C players spawn on top of each other.
- **T**: Same as above but for Terrorist (attackers)
- **P**: Area where you can plant bomb (free to walk around it like a blank space, but allows us to see where in the map the bomb can be planted. (it cannot be planted outside of these spots.
- **1**: Same as P but just to help indicate that this is A site in the map file. 1 site is the smallest amount
- **2**: Same as P but just to help indicate that this is B site in the map file. 2 sites is average amount.
- **3**: Same as P but just to help indicate that this is C site in the map file. 3 sites is the largest amount.
- **n**: This actually works the same way as walls but is meant to be areas where player cannot even touch. The idea is simple, that we can save memory but not allocating some of the characteristics, but so far I didn't set this up in the game. Still use it in the map file.

Next are some characters that show up in the game but not in the map file

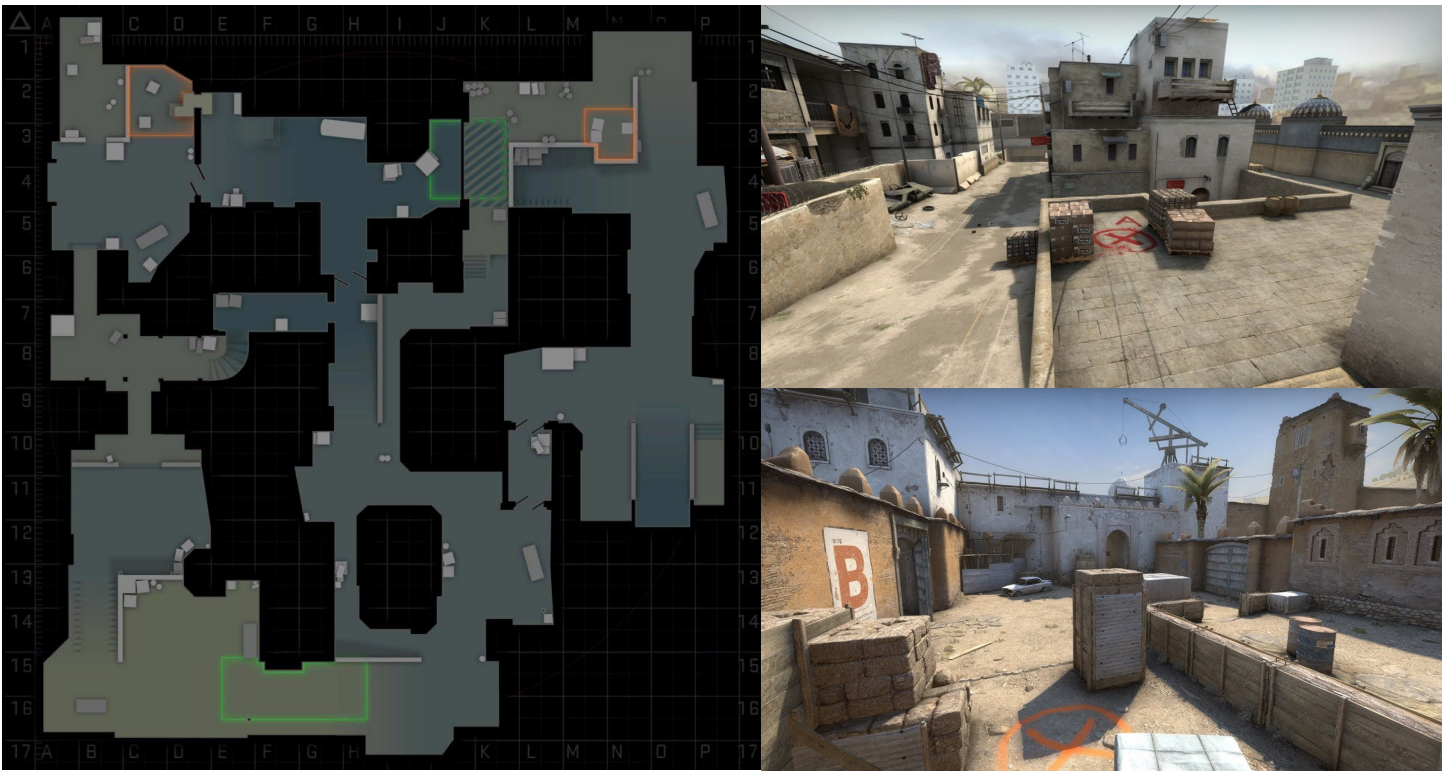
- **@**: Human Player (so the player that you control). So, we can keep track of where we are. We don't show this on the map input file since the spawn location for player is going to depend on the team he selects.
- **\***: The asterisk is used to show bullets as they are fired from a player's gun until they hit a wall or a player. Players die instantly from 1 shot. If you remember Goldeneye you can think of as all players spawning with the Golden Gun!

As you can see with this character set we can denote a well defined map. Some general guidelines for creating maps that help us not worry about weird scenarios are:

- The entire map is walled off so players cannot fall out of the map.
- The map has at least 1 bomb site and exactly 1 bomb that is usually near the T players.
- Usually the CT players spawn near the bomb sites whereas the T are further away allowing the CT players to have a few seconds to setup defensive positions before T's arrive if they were to rush straight to a site. Rushing B site is a world class renowned strategy invented by Russian players for winning this map. Usually denoted by the tactical communication, "Rush B, No Stop." Many world championships have been won with this 1000 IQ play.
- Map should be balanced by having more than one route to a site and allowing different ranges of gameplay from close range gunfights to long range gunfights.

Just for reference, the ASCII map I created by hand and out of memory is based on the world famous CS:GO map DUST\_2. It is probably the most played, balanced, and popular map that even non-CS:GO players have heard of. If you are curious here is an image of the real version of this map as it may help you better understand my ascii version of it along with photos of what it looks like in-game: (next page)





On the left you can see a map of the real Dust\_2. The green squares are the spawn areas and the red squares are the plant sites. On the top right you can see what A site looks like, and on the bottom right you can see what B site looks like.

Next let's talk about user input:

The bots will control themselves, but to control the player you will use a set of predefined keys. Make sure they work whether the CAPLOCKS key is on or off:

- Movement: WASD and arrow keys to move up,down, left, and right. Basically, have both set so user can pick whichever they want.
- To fire your weapon use: Space Bar
- You want to be able to quit the game at any time by pressing the letter Q
- Pressing C will clear the screen which allows you to re-render the game. Not useful in the game, but useful for debugging and fixing weird graphic glitches. So I suggest you implement it and leave it there for just in case. Along with Q you want to be able to refresh screen or quit your game at any time because ncurses library locks down the input meaning CTRL-C won't work and you will have to close the terminal window to close the program or in the case of sally, log off and on.
- To idle in place use the letter "i". The reason you would want to idle is if you don't have asynchronous input on and you want the game to tick while you stay in place.

Let me briefly talk about the ticking while I am at it. So by default when you do getch() on the terminal or anything like cin. It halts the program execution and waits for user input. This means you cannot keep running your game. This is an issue if you want the game to act like a real game in the sense that whether or not you are moving the game can go on. A way to solve this is using asynchronous input which creates a thread that deals with I/O so that the code can still be running and simply leave a piece of your program occasionally checking a buffer for any input a user has entered. That way if you enter anything the next time it checks the buffer it will then make the move. Feel free to implement this if you wish to, but you do not have to.

Next let me devolve deeper into the NCURSES library and some very useful information on how to use it:

First, I present you with the following two functions I made:

```
void initCurses(){
#ifdef curses
    // Curses Initialisations
    initscr();
    raw();
    keypad(stdscr, TRUE);
    noecho();
    printw("Welcome - Press # to Exit\n");
#endif
}

void endCurses(){
#ifdef curses
    refresh();
    getch();
    endwin();
#endif
}
```

Call the first function at start of the program and then call the second function at the end of the program. Let's look deeper into them. Notice the `getch()` function. This function will halt user input by default and request a single letter press. It will then return that value as its return value. This is much better than using `cin` because `cin` requires us to give input and then press enter. NCURSES does not and immediately reads the keyboard input. This is perfect for playing a game where we want to just press W to move up without having to press enter after it. This would be unplayable, literally.

Next, let us see the O part of I/O. Which means how do we print stuff to the screen? For that we use `printw` to print any output. The function works the same way as `printf` does in C. If you have never used `printf` then today is your lucky day! Go to: <http://www.cplusplus.com/reference/cstdio/printf/> and learn something new! 😊

Next, notice the `refresh` function. `printw` should flush the buffer out when you do a linefeed. But if you want to force it out – meaning halt the program, wait for OS to confirm output is printed, buffer is empty, and then keep going – similar to `cout << flush`, you can use `refresh`. I have seen issues on sally (not on my local Linux VM) where calling a lot of `printw` statements for each character of the map will cause issues where not everything is printed correctly, or at all, if you have these issues call `refresh()` after each `printw` statement and that should fix the issues at the cost of a little bit of lag when running the game.

There is a lot more you can do with `ncurses` but that is all I will talk about. There are many resources online to make the text very pretty and to do the asynchronous input I was talking about. Feel free to read those, use them, and cite them in your report. Also post on discord anything cool you find. I'd love to hear of any cool things that may make the game look prettier. As a started experiment with the color of the letters!

A word of caution with `ncurses`. Do not mix standard I/O with `ncurses`. In other words, don't put `cin` and `cout` statements mixed with `printw` and `getch`. Your program will begin doing weird things and may randomly seg fault. That is because `ncurses` library is meant to replace the `iostream` library, and not complement it. You have been warned!

To use ncurses library make sure you add the following include:

```
#include<ncurses.h>
```

and when compiling add the -lncurses flag to your arguments. For example:

```
g++ ast12.cpp -lncurses -std=c++11
```

Next I'd like to discuss the ballistics engine of your game

You will need to develop a mini engine that can handle ballistics in the game. Now before you panic this boils down to these basics:

- When a user fires their gun it:
  - creates a projectile entity that starts at the user's current location where he is shooting.
  - marks the owner of it as the user who shot it (so players don't kill themselves)
  - And then every tick of the game it moves in the direction it was originally fired
    - This direction is based on which way the player was facing when he fired it.
    - You may make it move 1 coordinate point as I did or try more than one for faster bullets. But if you do that make sure you check all points it passed through!
  - Each tick it checks to see if the location it is currently at is populated by an entity that it can kill.
    - So in other words if it has hit a player (friendly or foe since friendly fire is on!)
    - If the entity is not killable (such as the bomb) it just whizzes by and continues its direction.
  - Eventually because of game design the bullet will encounter a wall. In which case it will 'hit' it in the sense that the bullet will despawn (be deleted from the game... and don't forget to despawn it otherwise memory leaks will grow the more people shoot. I have seen this happen in real development lol.
  - Note that obstacles are not entities but are map marked in map with an o, bullets can just whiz past these and keep on going as well. Same applies for P symbols
- To make things easy the bullets are one hit kill and you need to make sure the engine alerts the game that the players have been killed so it marks them as such. The damage drop off is non existent and bullets will continue until they hit a player or a wall.
- You may have fun with this so if multiple players are in the same spot it can kill more than 1 player for those epic double kills 😊
- So I guess you could say even though it's called ballistics there's very little actual ballistics other than keep track and render your bullets with asterisks \* as they travel the map.

Just as a reminder, each tick of the game is measured either by a second of the computer if you are using asynchronous input, or by a click of the keyboard that sends a valid move to the human player (dead or alive, dead humans just ignore the input). Note that if you give invalid input it does not count as a tick and will keep requesting valid input.

Finally, I would like to discuss Skynet... I mean the Artificial Intelligence also known as the bots in the game.

You want the bots to act like real players and follow the same goals of the game to get the dub for their team. In fact, for the version of the game that has no human. You want it to be a cool showdown between the AI that is both skilled but not the exact same every time. It would be no fun if it always did the exact same each round. So while there will be a certain level of intelligence in the AI, you also want a certain level of randomness in their

choices (so a chance of error or a change in its goals) allowing each round to feel different and interesting. In short make it look like its players and not bots. Easier said than done...

Here is a list of their objectives that you must implement to help achieve these goals each AI needs to be able to do this each tick (Note I use the pronoun we but I am talking as the bot):

- **Run ViewFinding** to see if any enemies are in our Line of Sight Up, Down, Left, or Right. If they are shoot them.
  - NOTE: Technically we are facing in one direction so before we shoot we need to change our last direction to where enemy is and shoot. Otherwise bullets will not go in the right direction in 3 of the 4 cases. But that may break tunnels so we have to check if we are in a tunnel/bridge (#) and if so then we cannot change direction to a forbidden direction to shoot.

After, whether we shot or not we:

- **Run PathFinding** to see where to move and then make move.

We decide that based on team and certain factors:

#### **CT Bot (Defender):**

- If bomb is not planted we move towards one plant site (RNG 50% chance).
  - Use RNG within the site to not all camp on the Ps.
    - Instead make it so they have to be anywhere from 2-20 squares away from the bombsite they picked . Tweak these values as you see fit.
- If bomb is planted we move towards the bomb to defuse it. (if we see enemies, we will shoot at them each tick because each tick the ViewFinding would be running before the pathfinding. Since our goal is to get to the bomb it will look like we are shooting as we run to defuse)
  - RNG Should be added here to make mistakes and/or to focus on killing vs bomb defuse.
    - In short, it's possible the bot decides to go Rambo and get kills instead of defusing. Just like a human usually does. Plus, defusing while we are being shot at usually ends up in us dead.
- Finally, if bomb is not planted and one of our teammates sees an enemy we rotate towards last seen location of enemy. Rotate means leave your bomb site and go towards the enemy. This typically happens in real games since there is a higher chance that is the site the enemy will attack. Of course, there can be ruses and whatnot, but at some point, we have to rotate before we lose the sight...

#### **T Bot (Attacker):**

- If bomb is dropped (as in it is laying around) pick up bomb.
  - Add a 10 tick delay at start so human players have a chance of getting it before the AI does.
  - If there are no human players in game, do not add this delay.
  - There can also be a RNG that they don't want bomb and go towards random bomb site instead even after the 10 ticks. Just like real players will do.
- If bomb is picked up by a teammate you go towards bomb carrier.
  - Still keep the RNG I mentioned above of having some bots choose to go do their own thing like going to a different Bomb site without bomb looking for kills Rambo style.
- If bot has bomb, he goes to a Bomb site (RNG to pick which one) to plant bomb.
- After bomb is planted all bots (regardless of previous RNG) go to bomb site to defend.
  - Those already there do the same logic as CT in start: that is to camp bomb site but not on top of the bomb or the P area, but around the area within a designated RNG range (2-10 or so). That way they are spread out.



- In all cases above again by default they can shoot and run since viewfinder runs each tick so they should be able to defend the site.

Doing all of this will make the AI be able to self-play the game and hopefully be challenging to a human player if there is one.

To compute a location that the AI has to move to you will use Dijkstra or other Pathfinding algorithms of your choice (but you must code them from scratch) so that it can know how to get to the destination. For this the bot can see the entire map so that it can create a path graph representing it and regenerate the Shortest Path after each tick update.

Make sure that each tick you check all of the win conditions describe in the first page of the assignment (team wipe, bomb exploding/defused etc..)

Also make sure you allow team select to happen when the game launches where they enter T or C for the team they want or a different key for not joining game and having a bot only match (I did the key "I").

That about wraps up the entire requirements. At this point you can jump to the closing notes on the last page of this document, read that, and then get coding! However, if you would like some guidance on where to begin, or how to implement certain requirements feel free to read on as I discuss my implementation. You may also watch the YouTube lectures on Dijkstra as I discuss the assignment there along with the 5/5/2020 lecture where I give a demo.

## My implementation:

First, I should point out that I tried to keep everything as Object Oriented as possible meaning I did not have any global functions or other weird mockery. At the end of the day it's best to look at this as an engine we are coding more than a game and that it may be part of a larger scheme of things and you don't want to run into naming convention problems. To that extend I want to point out that I do neither include the `iostream` library or use namespace `std`. Because we are using `ncurses` library we barely even have to do `std::` throughout the code aside from a couple of strings. This is not something I discovered either and is pretty common in real programming. As I write this I will try to keep unnecessary details out to try and make it as concise, yet thorough and useful, as possible. As always you can ask on discord and I can clarify any sections you may particularly need more details on.

I do have the following includes:

```
#include<fstream>           //Filestreams for opening map text file
#include<ncurses.h>          //NCurses Library
#include<stdlib.h>           //srand for adding RNG to AI
#include<time.h>             //time for seeing RNG
#include<string>             //to_string c++ 11 function
#include<vector>             //vectors, obviously, lol.
```

Next let me give you a brief overview of the classes I have and discuss what each one does. I am listing them in the order that I declared them in my code. I did do the entire project as a single file for convenience, but I recommend you do split it up as my implementation is well over 1,000 lines of code:

- **CharMap** (you will see in the Skeleton Lite that this class has a 2D char array that stores the map read from the text file in the constructor of the class, and also has a function that can print it. I don't actually print it ever as I will read this data into different classes but it's mostly so I can just get the data from a

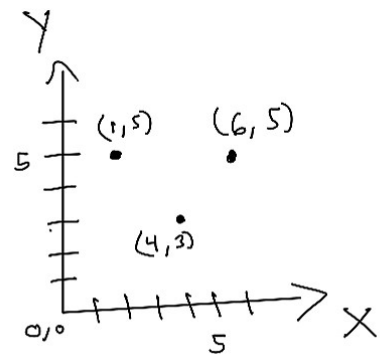
file in hard drive into RAM where I can process it faster. It's the equivalent of doing all file loading in the loading screen of a game versus loading things as we go which can make games stutter even in modern computers. Look no further than GTA V to see this phenomenon when it is installed in an HDD vs an SSD.


- **ent\_t** This is a very important super class of a couple of follow up classes. It stands for entity type and is an abstract class (to refresh your mind, abstract class means there is at least one pure virtual function and at least 1 fully defined function – unlike the term interfaces in C++ that usually denotes classes that only have pure virtual functions). What makes it abstract is that I have a pure virtual function named `whatamI()` that returns a char depending on what inherited object it is. I'll talk more about that in the next class.
  - I also have a `setCoordinates(x, y)` function that allows me to set the class variables `x` and `y`. These variables allow the entity to know its position in the game level/map which is very useful to handle a lot of things involving movement of bullets and players (AI or Human).
  - I do have an ID system setup using a static counter so each entity has its own unique ID. I don't actually use it, but I thought it would be useful to have for debugging purposes or simply to be nicely organized.
  - It also stores the `creationTime` of the entity which I also don't use but is nice to have.
  - There is a print function that prints out details of the entity (location, id, creation time)
  - Note that the constructor requires `x` and `y` coordinates this way entities have to have a location when they are created.

I'm going to take a moment here to discuss the coordinate system of the entire game. I have `x` and `y` values, but it does not work like a normal coordinate plane. Because of the way the screen renders. It is as if we took a regular `x/y` plot and we tilted our head 90 degrees to our left:

As an example, suppose we have a map that is 20 units high by 50 units wide. The following statements would be true of such example:

- The range of `x` would be 0-19.
- The range of `y` would be 0-49
- Coordinate 0,0 is the top left of the screen and the first character we would print.
- Coordinate 0,49 would be the top right of the screen.
- Coordinate 19,0 would be the bottom left of the rendered output.
- Coordinate 19,49 would be the bottom right of the rendered output.

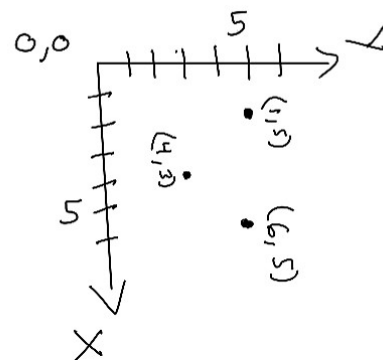


Tilt head left 90°   
or rotate image clockwise (right) 90° ↻

It's important that you take a minute to understand this. Otherwise when you are calculating movement you will get confused and move in unexpected directions. As a rule of thumb assuming you have the map stored in a 2D array like `arr[x][y]` then do:

- Up: `arr[--x][y]`
- Down: `arr[++x][y]`
- Left: `arr[x][--y]`
- Right: `arr[x][++y]`

Now we can continue talking about classes 😊



- **bomb\_t** This is derived from the `ent_t` class. Because of it, we need to define the `whatamI()` function that we discussed earlier so we can instantiate an object of this class which represents the bomb. In this case we define (so hardcode) it to return the char 'B' to represent the class. The advantage of having all of these derived classes from `ent_t` is that we can keep a vector of entity pointers that point to different derived objects achieving polymorphism and making our lives easier. If we then need to identify the type of entity it is, we can check what the `whatamI()` function returns and then dynamically cast the pointer to the derived class so we can access the specific functions and class variables that the class contains. In this case those are just a couple of self-explanatory Boolean variables that tell us the state of the bomb. I initialize all of them to false in the constructor.
  - **isPlanted**
  - **isCarried**
  - **isDefused**

A quick reminder, because `ent_t` class expects a `y` and `x` to store the location. Make sure your constructor passes those variables along. Here is an excerpt of my code to remind you how to do this.

```
class bomb_t : public ent_t {
public:
    bomb_t(int xx, int yy) : ent_t(xx,yy) {
        isPlanted = false;
        isCarried = false;
        isDefused = false;
    }
}
```

- **projectile\_t** This next class also is derived from the `ent_t` class. In this case we use char '\*' to identify it. The class represents a projectile entity. This means a bullet that a player shot in the game. The fact that the entity exists means that the bullet is currently flying in a specific direction that we base on the direction the user was when he fired the gun. Because of that the class contains a `char` variable called `direction` that stores that direction/momentum. It also has a pointer of `ent_t` type called `owner`. This keeps a pointer to the player entity (we will talk about that next) who shot the projectile this way we do not accidentally shoot ourselves.

The real logic behind this is because when a player shoots a projectile, it gets spawned in the same coordinate as the player that shot it. Because of this, it is technically *touching* the player which means it is technically hitting us. Imagine you were at the gun range and were shooting a pistol. Normally you hold it away from you in your hands (pointed in a safe direction, of course) and you depress the trigger and the bullet gets fired towards your designated target. Well in the game because each coordinate is rather big, one full block where an entire player, or more resides, that would be the equivalent of sticking a gun in your center of mass, and firing it out towards the target. You'd essentially be shooting yourself in the process. This is a very typical bug for amateur game developers to make which is why I am pointing it out.

To expand on this a little further and go off topic, this is also a pain when it comes rendering bullets in real games. Normally bullets spawn from your coordinates, and assuming we are already not shooting ourselves, the bullet's trail or tracer (the sfx that the bullet makes showing it's trajectory in a lot of games) is rendered off of the location of the bullet. So to other players it looks like you are shooting the guns from your belly and not your pistol. Not exactly a very realistic game there... Ok so you fix that by making the bullet spawn from the location of the muzzle of the gun. Great now it looks fine to others, but it creates another problem: When we shoot a real gun we bring the sights to our eyes blocking a lot of our vision. This would be similar to Aiming down the sights in games like Call of Duty. That's okay

no problem there, but what if you fire the gun while not aiming down the sights? In most games the gun is still visible in the screen when shooting and not aiming down sights, when in reality your model is holding it far lower around the hip area. This should not even render the gun in the screen, but then the game doesn't look as cool and we can't get away with selling as many gun skins, so we render the gun anyways. In what is essentially an unrealistic angle. Don't feel scammed. Hollywood does this all the time with camera tricks. That's where video games actually learned it from.



**Example of tracers both coming and going.**

Well now we have a problem. Do we spawn the bullet off the real location to others or the one that we see in FPS model? If we do what is right for others then the bullet tracers will not look like they are coming off the gun for us, and the aim will be totally messed up because even though we are aiming down the sights, the crosshair should be aiming to where the bullet is firing. But the math doesn't add up. On the other hand, we can line it up so it shows the tracer in FPS correctly to us, but then the bullet is going to be spawning near our face to others along with the tracer? So what do we do?



**Example of where gun shows for other players versus where it shows for the player in FPS mode.**

There are a lot of solutions but a simple one is for long range shots it makes little difference so render two tracers: the tracer for the user as if it had come from the face and have the bullet's math computed from the correct spot, then for others render the correct tracer that the bullets math generates. For close range shots then this is the part where we mess with the users cross hair to show what he is really aiming at isn't what is being aimed at and we try and create a fake tracer that will still hit in the position the bullet will hit.

Just a nice aside to point out that even in such a basic game we have to consider weird logic like our own bullets shooting us when we fire them. For the render side the only thing to note, and we will



discuss later is that we always prioritize rendering the player over any projectiles that reside in this coordinate point. That way we do not risk having the player disappear from the map for a tick when the bullet gets spawned.

Anyway back to our game we only have to worry about the bullet killing us when it spawns so the owner pointer to the player that shot can be checked later on to ensure we don't consider us touching the bullet a hit. Again, because this is a derived class we pass the `ent_t` class the `x` and `y` coordinates of the projectile which initially will be those of the player.

- **player\_t** This class is also derived from the `ent_t` class. It represents a player entity whether it is a human or a bot. It does have a couple of class variables worth discussion:
  - **isHuman** Helps to differentiate if player is a human or a bot.
  - **bomb** This is a pointer of `bomb_t` type and is set to null unless the player is carrying them bomb. In which case we save a reference to it. Pointer magic at it's best!
  - **lastDirection** Useful to know which way we are facing so if we shoot projectiles they fire in that direction. Also if we are in tunnels we can only move parallel to our last direction and not perpendicular so useful to have for that hack.
  - **status** tiny enum that keeps dead or alive. Could just be a Boolean or could be renamed to `isAlive` and do the same thing. Useful for AI to not have to do any pathfinding for a dead player.
  - **Team** a char that keeps 'C' if we are in CT team and 'T' if we are in T team. It's very important to associate a team with a player since CT can do things that T cannot do such as defuse a bomb whereas Ts can carry a bomb and T's cannot. The disadvantage to having one class of players versus two is that CT do not need a `bomb_t` pointer but it's a small price to pay to not go over the board with the number of classes. Were this a more in-depth game were the a team has different subclasses like overwatch, you'd definitely want to split this into multiple classes and make `player_t` abstract and have the derived class be the one that you define. In fact you'd probably also want human players to be a derived of this class with all of the other features humans have that bots don't. That reminds me, the identifier for this class's `whatAmI()` function is going to be T, C, depending on the team.
  - Notice I do not have a Boolean for whether player is carrying bomb, that's because I can just check if `bomb` is not equal to null.
  - **void RIP(std::vector<ent\_t\*> & p\_entList)** This function runs when the player dies by either a projectile or if the planted bomb explodes and he is within range (notice I never mentioned this earlier, that's because it's optional if you wish to implement bomb explosion damage). This function needs the point where player died (we will talk about the point class next) but since we have not declared `point_t` – as that would cause a circular dependency – we will work around this by instead just passing a reference to a vector containing pointers to all entities that are currently in that point where we have to remove the player from and add the bomb to. I'm not proud of this function, but it works and a better solution would require refactoring code so we will leave that for another time which in game development terms means probably never. I actually had to think for a minute on this function because of the circular dependency risk. Sometimes you want a class to have a reference, or pointer in C++ terms, to another class. Then you want the other class to have a reference to this class. In your head it all works out. But the problem is the code cannot resolve incomplete types of the second class you write code for, and while a class forward declaration may help, in reality if you are doing something that smells like a circular dependency then you it's a sign of a deeper problem in the structure of your classes. In my case here all I needed was a vector of `ent_t` pointers. Passing an object of another class just for that causes more issues than it solves and therefore is the right choice. Were I to recode this, I would not have this `RIP` function here but somewhere up the hierarchy. For now make sure you check if player was carrying bomb and if so move it from the player's `bomb_t` pointer to the `point_t`'s `entList` & change `isCarried` to false on it.

This concludes all of the entity class derivatives. Next is a very important class:

- **point\_t** An object of this class represents a 1x1 grid point and what is contained within it. It also represents a vertex in the graph we use for Dijkstra's Shortest Path! What a coincidence... This class is the heart of the game because it manages and renders that grid. The level class we will talk about later keeps a 2D array of these points the same size as the CharMap array, but this one is where everything is rendering from. First let me start by listing the class variables which will be explained as we move along the different functions this class has:
  - **isBombsite**
  - **isObstacle**
  - **isWall**
  - **isBridgeTunnel**
  - **baseType** A char that keeps the base type of point. In other words while entities may come and go through the point. This remains the same. For example. if the baseType is a ' ' then that means that you can walk here and while players may walk over it, which then shows the players in the map, it still needs to remember what it was so that it can reset to the default ' ' after the entities leave.
  - **x**
  - **y**
  - **id** Point have its own ID system again, I never used it either...
  - **std::vector<ent\_t\*>entList** Active entities that are currently residing here, like players.. Note that it keeps pointers to the entities, not the entities themselves. In other words this is empty if no entities are here, but as they are spawned here or they move here we add a reference to them here. Then when they move to the next point we copy the reference over and then delete it from here. If a player spawns here we actually create using new the player and we have the only reference to it, but eventually it gets passed to the AI or other points, so take good care of the references. This is a very important vector and I have over 60 references to it in my code.
  - **Constructor:** The constructor if this class is going to take the character from the char map of the same location in the respective 2D array along with the x and y coordinates it lives at. Just like it is useful for our entities to know where they are, it is also useful to know where a point is and makes some of code cleaner as long as you don't forget to update all of these x and ys... In retrospect I would have made 1 pair of x and ys and keep references to them in all of these classes. So, the first thing we need to do in the constructor is parse the character from the CharMap and fill it with entities in special cases. You can refer to the above key on the types of characters we may get. As an example if the CharMap contains an x we want to set the baseType to 'x' and set the isWall boolean to true. This way when we try to move a player here we can reject the move after seeing it is a wall. Or if a bullet lands here then we can despawn it. Some other types and how we handle them are
    - x,#,o, blank space ' ', and n set the baseType to what the char is.
    - B: this is a bomb so we want to set the base type to a blank meaning anyone can walk here. That way they can pick up the bomb and then point can become a normal blank space. We also want to spawn the bomb here by adding it to the entList. We can achieve that with a single line of code:

```
entList.push_back(new bomb_t(x,y));
```

- P, 1, 2, and 3 you can set baseType to ' ' and set the isBombsite to true to remember to render a P.

- For C and T (players) we create 5 players per team. Yes we will eventually delete one of them if a human joins, but that's okay and will happen elsewhere in code. That's actually how it works in CS:GO as well. If bots are enabled they will spawn and then get kicked the second a real player joins. So why question Lord Gaben?
  - **renderPoint()** This function decides what is printed for that coordinate. We start with the basetype as default, but then we check what is in the point's `entList` and if there is anything, we print some of those instead such as the Bomb, players, projectiles, etc... The only trick to this function is we need to give priorities to what gets printed, Such as players over Bomb over an empty spot. This is where we solve the other half of that projectile issue by ensuring players always print before projectiles do. The priorities that looked good from highest to lowest were: @, T, CT, \*, B, P and then everything. However, when coding I actually coded them backwards so that I first set the `finalType` (a local function variable that is what I ultimately `printw`) the `baseType` of the point then if it is a bomb site I set it to P, then if there is a Bomb there I set it again to B, then if there are players I set it to the team, and so on until I get to the human as top priority. That way even though it's not the most efficient. It will keep overwriting with a higher priority until it cannot find one with higher priority. If you wanted to be fancy, instead of a vector here you could keep a priority queue and then you could always just peek the head node to know what should be rendered. Not only would it be more efficient, but it would be pretty neat!
  - **deleteEntFromPoint(ent\_t\* e)** This function is pretty straight forward and just deletes an entity pointer from the `entList` vector if found. The only reason I decided to include such trivial function is because I actually added it near the end of development after I found myself copy pasting the delete code in a few places and realize I really should have made it a function from the beginning! If you are curious I am at about line 300 of my source code so far.
- **Level** This class is keeps all the points together ( so 2D array) and is the META lol. In other words it keeps everything together and is what I actually deal with in main. It's the heart of the 3-way engine with the other 2 components being the AI engine and the Ballistics Engine (so our poor man's physics engine). First let's talk about the class local variables it contains:
  - **point\_t\*\*\* point** 2D array of pointers holding all points. This is the same dimension as the `CharMap`. The reason you see `***` is because each point is dynamically allocated. In C++ you cannot call non-default constructor when you create an array of an object to get around this, I make an array of pointers I then manually (in a loop) allocate using `new` one by one. Easy workaround to a C++ limitation.
  - **height** First dimension of 2D array.
  - **width** Second dimension of the 2D array
  - **CharMap\* mapref** a reference to the original map. Useful to have just incase things go really wrong.
  - **roundTimer** keeps the number of ticks before round ends. Used for checking endgame condition. Yes we're in the endgame now, lol
  - **bombTimer** countdown after bomb has been planted.
  - **bombPlanted** Useful to know if bomb is planted. Yes I could also use the above but it's cleaner this way.
  - **Talive** How many T players are alive
  - **Calive** How many CT players are alive
  - **endCondition**
    - 0: Not Over
    - 1: Bomb Defused, CT wins
    - 2: Bomb Exploded, T wins
    - 3: CT dead, T wins
    - 4: T dead and bomb isn't planted, CT wins

- 5: Time is Up and Bomb isn't planted, CT wins.
- **Constructor** This takes the `CharMap`, and creates the 2D array of pointers. It makes sure to pass the character to create it with along with the coordinates. It also initializes variables mentioned earlier to 300 for `roundTimer`, 30 for `bombTimer`, and 5 for both `Talive` and `cAlive`. The rest are trivial.
- **renderLevel()** This is one of the few functions we call on main. It takes care of rendering the latest version of the point array by calling `renderPoint()` on each point.
- **clearScreen()** Another function called by main that will clear the screen in preparation to calling the `renderLevel()` function. It also prints out some stats like whether bomb is planted and time left to the round. I could have copied the code from this function to the beginning of `renderLevel()` but it's nice to have it separately so it can easily be commented out and then we can have a log of each frame of the game for debugging and grading purposes 😊
- **openNteamSelect()** This function starts the game and asks player what team he wants. Then it creates the human player, assigns it to the right team, removes one of the AIs on the human's team to replace with player so it is still 5v5, and returns a pointer to the human player so we can manipulate it in main. Which boils down to being able to control his movements. The only reason the human player is not kept in level as well is because I wanted ability to add local multiplayer at some point. You know, for the paid DLC. Make sure you call `renderLevel()` to start up the game at the end of the function.
- **mapSearch()** Returns the x,y coordinates of the first instance of a type searched for the map (not level). Useful for searching for T or CT spawn or Bomb's spawn location. In my case I use it to find out where to spawn the player based on team he selected.
- **secondTick(), bombTick()** both of these decrease their timers each tick of the game. Whenever you hear me talk about tick, that's the `secondTick()` function. It should only be called once per render or second if you are doing asynchronous input. If you want to increase the tickrate to faster than that you can but the game will run faster. Basically it is how often it recomputes things, google server tick rates if you are curious about this stuff.
- **checkRoundStatus()** This function also gets called from the main loop to make sure game hasn't ended by checking the class variables for things like, bomb timer not being 0 (bomb exploded) or teams having been wiped. It essentially checks all the local variables and updates as necessary. It also returns the status so the main can see if it's game over. The last thing this function does is increase our `secondTick()` and should be the last thing in our main loop before we start over.
- **Destructor** I haven't been writing much about these so far, but all classes should have them. In this case I call delete of each point, but make sure you take time to do all of them as otherwise you are going to end up with so many memory leaks people may think you are a real AAA Game Developer!

Now we are about 450 lines into the code and we finished with the first engine and can talk about our poor man's physics engine class:

- **BallisticDispatcher** This class handles all of the projectiles that currently exist in the level. Therefore it contains two class variables:
  - **Std::vector<projectile\_t\*> projList** Contains pointers to all active projectiles
  - **Level\* levelref** Reference to level. Useful for accessing and updating level with ballistic information. It is key to realize that this must be declared after level because of this meaning that it is a one-way road. Level cannot modify ballistics. That is okay because only AI and the `MovementDispatcher` via main can add projectiles using the **addProjectile(projectile\_t\* proj)** function. Which just pushes to the vector.



- **updateAll ()** is where all the magic happens. This function is called by main each tick and updates all projectiles location by 1 coordinate point. So they move one box. It also checks if any projectile has collided anything and handles that. The logic works by looping through each projectile in the `projList` vector:
  - Find where projectile is moving based on the direction. By this we use the level pointer to find the point that they are at and the one that they have to move to.
  - Once we have that point we check to see what is there:
  - If it's a wall, then all we need to do is despawn it, no need to actually move it to the wall. We simply call the `deleteProj` function and pass the point and projectile pointers. All this function does is remove it from `projList` and also call `deleteEntFromPoint` in the `point_t` class so it gets removed from the point's `entList` where it was residing. It's important that we do the latter so it actually gets removed otherwise it will continue to be rendered forever right before it hits the wall! For debugging purposes it's a good idea to see if the delete succeeded with a Boolean to detect any bugs. Do note that by delete I mean removing the pointer, we still have the memory allocated and in any of these deletes you need to make sure that you call delete on it only after all references are removed to it to avoid dangling pointers. Also be careful of deleting memory twice as well. My recommendation is that any actual de-allocation happen in `point_t` and not in the Ballistic Dispatcher.
  - If it's not a wall, then we check to see the size of the point's `entList`. If it's 0 then there is nothing at destination and the projectile can move there. To do this we:
    - add a pointer/reference in the destination point's empty `entList`
    - Update the projectile's coordinates (remember all entities have x,y coordinates)
    - Delete the projectile pointer from the old point's `entList` since it is no longer there by calling `deleteEntFromPoint`
  - On the other hand, if the destination `entList` was not empty we need to loop through the list to see what is in the destination point and handle each entity depending on the type:

```
for(int i=0; i < to->entList.size(); i++){
    if(to->entList[i]->whatamI() == 'T' ||
       to->entList[i]->whatamI() == 'C'){
```

- As shown above if the entity is a player, it will have the `whatamI ()` set to T or C. That means we hit a player. Before we declare it a kill though we want to make sure that it's not the owner of the projectile. Remember that whole spiel about that? Yeah that's why. So we can just compare that pointer with the `owner` pointer in the projectile for that.
- Once we have confirmed it is not ourselves. Then it's a real kill and we can call the `RIP` function in player. To do that we dynamically cast the `ent_t` to `player_t` and then call the `RIP` function.

```
player_t* player = dynamic_cast<player_t*>(to->entList[i]);
player->RIP(to->entList);
```

- Note that we are passing the `entList` of the point where the player died so the `RIP` function can handle dropping the bomb if the player who died was carrying it.
- Also note that we do not need to delete the player from `entList` as the `RIP` function manages that.
- The `RIP` function also sets the `playerStatus` to dead. This is important because `BallisticsDispatcher` does not have a way of updating the AI Dispatcher that a player is

dead directly, and the AI dispatcher needs to be able to track all of this information for the bots to work, but what we can do is that once `RIP` sets the status to dead, the next time the AI Dispatcher runs it, as part of its routines it will check the status of all players, then it will see that a player has been marked as dead and will stop tracking him by deleting it from their player list. We will discuss this more once we get to the AI Dispatcher, but what I do want to emphasize is that the reason we cannot update the AI Dispatcher object directly is because at this point the class has not even been declared. And the AI Dispatcher does have a reference to `Ballistics`. Since that is far more important for the AI to be able to shoot. So, you may ask, why not add reference to both classes of each other. That's again a circular dependency and it's very bad as in your code won't even compile. Similarly, `player_t` was declared before `point_t` and therefore it cannot have a `point_t` as a parameter to its function. This is why we can only pass an `entList` but that is okay as we don't need anything from `point_t` other than the list of entities. So this is both efficient and avoids circular dependencies.

- Lastly, don't forget to despawn the bullet after we hit someone unless you want it to keep going afterwards for those lineup killstreaks. Do wait to check the remaining entities in `entList` to see if more players were in the same coordinate and should be killed as well. That does remind me of an important point. Look at the code on the previous page, it's a for loop that runs from 0 to the size of the entity list. There is a risk of skipping entities in that list if you just keep looping like normal after deleting something. Even if you didn't delete it explicitly, as previously mentioned, `RIP` deletes it. So, when we come back from the function, there is one less entity and the size is smaller. The problem is that if the entity we deleted was in index 3 and there were 6 entities, post delete there are 5 entities but the entity that was in index 4 is now in index 3 (That's vectors for you). However, the next spot to check is index 4 (which now actually hold index 5. This means you skipped an entity entirely and it's hard to see because it didn't even happen in this code's function! To fix this you have to manually decrement your loop counter `i`. It is a bit of a hack but Modern Problems Require Modern Solutions. Also this is why it is important to call the size function in the loop and not before so if the size changes your loop can end earlier or later than originally planned and avoid a potentially nasty out of bounds error!
- Note that not all entities are killable. It's possible we found the bomb but we do not want to hit it, we just want to keep going and do as we did before by deleting the `proj_t` pointer from the old `entList` and adding it to the destination's `entList` on the adjacent point. At one point I had a bug where the bullets would stop in front of the bomb waiting for the bomb too move before continuing because they had hit some entity that was indestructible and it wasn't updating their position. For a second I thought the bomb was Neo!

That about covers all there is about the Ballistic Dispatcher. Now we are around line 550 of our source code and move on to talk about our Movement Dispatcher.

- **MovementDispatcher** This class is just a container for three static functions. They could be global functions but it felt nicer to keep them together and slap a cool name on it. They all deal with movement and user input. Basically the process of either reading keyboard input and converting it to movement by a player, or also converting movement sent by the AI Dispatcher into actual movement in the level. The three functions are:
  - **static char readkeyinput()** This function Reads Arrow Keys and also WASD/wasd for player movement. It's essentially a big switch statement that can handle upper and lowercase letters. It can also handle other options I already mentioned such as pressing 'Q' to quit and 'C' to clear screen and 'I' to idle. Small tip. To read arrow keys use: `KEY_UP`, `KEY_LEFT` and so on. The function is recursive by nature because if an invalid key is pressed it just calls itself again hoping a valid key is pressed this time. The function returns a char of a list of acceptable chars we can then parse. Basically this function simplifies the input so we only need to deal with

‘u’ for up and not KEY\_UP or U and also removes garbage user entered. It’s best to do it all in 1 function and not have to repeat these checks in 5 different places. If you are not using asynchronous input you will call this function in your main loop one time per iteration. So basically the game ticks only on valid input and otherwise keeps waiting for valid input. If you do use asynchronous input then this should be constantly open waiting for valid input.

- **static void makeMove(Level\* lvl, player\_t\* p, char direction, BallisticDispatcher\* ball)** This function is immediately called in main after readkeyinput by feeding the output of that into this one (direction). This function handles the actual movement of players in the map. It is also called by the AI Dispatcher to move bots. It’s a big function because it has to do checks of where the player is moving similar to Ballistic Dispatcher. For example if the player is trying to move to a point who’s baseType is a wall, then it will not make the move. You also want to make sure as that the player you are moving is alive. The AI is smart enough to not have dead bots move like scary zombies (DLC anyone), but the main sends any user input for the human player here after it is cleaned up in the readkeyInput function which means that if a human player is killed you want to ignore any rage input they enter into their keyboard! After all checks are done, when you are ready to move, refer to the earlier discussion on coordinates so you don’t mess up the direction you need to move entities and make sure that when you move the entity that you add him to the new point’s entList and then delete him from the old point’s entList. A useful reminder here are our ‘#’ tunnel/bridges. They only allow users to move parallel to the direction they were when they entered it. Make sure you account for that and limit movement in illegal directions. Lastly, don’t forget to update the entity’s coordinates with the setCoordinates function which just updates the x and y that way the entity is aware of it’s new location. Note that we can also generate projectiles here if the user presses spacebar, hence why the ballistic dispatcher reference is passed in. Similarly, we can do nothing if user passes an ‘I’ for idle.
- **static void postMovemenetChecks(Level \*lvl, player\_t \*p)** The last line of the makeMove function is a call to this function. It just made more sense to split the code since both functions combined are over 150 lines of code. Once you actually have moved an entity, this function does the any checks on the new location. The checks at the basic level depend on what team the player is. For example, if the player moving is ‘T’ team it needs to check if the bomb is in the point we just moved to and if it is then we want to pick it up automatically.

It also needs to see if the player moving already has the bomb and they location they just moved to is inside ‘P’ (so they just entered a bombsite), and if so you want to automatically plant the bomb. This is easier said than done for either of those cases, here is the latter as an example:

```
if(p->isCarryingBomb() && to->isBombsite){
    //Plant Bomb:
    p->bomb->isPlanted = true; //we planted it
    p->bomb->isCarried = false; //therefore we are not holding it.
    p->bomb->setCoordinates(to->x, to->y); //update where bomb was planted
    to->entList.push_back(p->bomb); //save bomb to point's entList so
                                   //it shows in map
    p->bomb = NULL; //cut off link to bomb since we aren't holding it.
    //Activate Timer
    lvl->bombPlanted = true; //Update Game that bomb
                           //is planted so tick starts!
    printf("Bomb Has Been Planted!");
    //getch(); //Just to pause execution to see if it works
```

That should give you a taste of how meticulous you should be for each of the checks/updates of this function. Similarly, you need to do a few checks for CT players so that if a CT arrives at the point where bomb is planted they can instantly defuse it (unless you add a delay). At which point you want to update the level's `endCondition` to 1 since that's pretty much gg.

That wraps up all of the Movement functions. Again these could be global functions but by keeping them in a class we make things more nicely packaged and we may in the future if we want to do movements stats or anything fancy we can remove the static keyword and create some class variables then just add an object in main of it.

This brings us to our last major class. The AI Dispatcher. At this point I've been writing this PDF for 16 hours... so I'll do my best to keep it coherent.

- **AIDispatcher** This handles the entire AI of the game and communicates with just about any class in the code. After all Ais need information to work. Here are the class variables:
  - **std::vector<player\_t\*> botList** A vector that holds reference/pointers to all Bots. It works similarly to how `projList` in the Ballistic Dispatcher worked.
  - **player\_t\* human** reference to human player entity
  - **bomb\_t\* bomb** Reference to the bomb entity
  - **Level\* levelref** Reference to the level so it can modify or access as needed be.
  - **totalBots** useful to know how many bots we started with since as bots die they are removed from `botList` entirely.
  - **BallisticDispatcher\* ballref** Reference/pointer to the ballistic dispatcher so we can call the `makemove` so Bots can move and/or generate projectiles (shoot stuff).

Next are the functions:

- **addHuman, addBot, addBomb** All these functions take the appropriate entities to initialize the class variables.
- **Constructor** I'll start by saying that we could make this easier with some pointer magic in other classes but this is the cleanest and safest way to do it. This function needs to search the 2D array of points in level for entities to add reference of in here such as the bomb, bots and the human player. To do this we just run through each `point_t's entList` and look for them. The reason for this is that the `point_t` constructor is the one that handles creation of entities. Nothing special about this search. We just use the `whatamI` function to check the entity type and if we find players we add them to the `botList` or the human pointer based on `isHuman` Boolean we made in `player_t`. We use a similar method to find and add the bomb. To make this more efficient we want to run through the 2D array only once checking each `entList` for any of the relevant entities. At the end of this function you want to also build the graph that will be used for shortest path. Do it as a separate function of your choosing.
- **printStats()** This function doesn't really do anything other than print log messages and is useful to call for debugging purposes. Basically the explains what the AI is doing why.
- **checkForNewDead()** Continuing my discussion on RIP earlier, we were not able to update the `botList` from there so now we can go ahead and update it by searching through all of the bots for any whose status is marked as dead. We do not need to do anything else as they have already been removed from the appropriate point's `entList` however we do want to de-allocate memory for it. Which hopefully for this case we did not do otherwise we are accessing dangling pointers. Same applies for checking if the human player died and if so we update that by setting it to NULL.
- **updateAll()** This is the function called in main each tick. This is where you have the AI do whatever an AI does. Does it swing from a web? In all seriousness I don't want to spoil your



imagination but look at the section I talked about what an AI should do and put all of that here in a for loop that runs everything for each bot in `botList`. I suggest that you split it into additional functions that you sequentially call here. An example of that is running `ViewFinding` and `PathFinding`. after which you call `makeMove` if the bot needs to move, or shoot. So you may potentially call it twice if it needs to shoot and move, or just once if it needs to move. You technically don't need to call it if the bot needs to just idle, such as protecting a bomb site, but for consistency I would call it with the 'I' character so it idles like human players do. That way you can print a debug message anytime idle is executed. I suggest you keep your Dijkstra's graph and shortest path computations on a class separately that is just queried here so you can compute the latest shortest path to the bot's objectives.

That wraps up all of the classes and all that remains is to discuss the main function which is rather short.

- **Main (client code):**

```
int main(int argc, char **argv){
    srand(time(NULL)); //Comment out to always have the same RNG for debugging
    CharMap *map = (argc == 2) ? new CharMap(argv[1]) : NULL; //Read in input file
    if(map == NULL){printf("Invalid Map File\n"); return 1;} //close if no file given
    initCurses(); // Curses Initialisations
    //map->print();
    Level* dust2 = new Level(map);
    player_t* player1 = dust2->openNteamSelect(); //Add Human Player
    BallisticDispatcher* ballistics = new BallisticDispatcher(dust2); //handles ballistics for the
    //game. It's cleaner if it is not part of level
    //Next we will create the AI Dispatcher that will handle the Artificial Intelligence (Skynet)
    AIDispatcher* AI = new AIDispatcher(dust2, ballistics); //This also adds all players
    //(human/bots) to AI so it can manage the pathfinding

    //Main Loop
    char ch;
    while((ch = MovementDispatcher::readkeyinput()) != 'q') { //Super important to not mess up
    //parenthesis here else major fail.
        //This next line is the one we'd need to do asynchronous if we wanted to go that route
        MovementDispatcher::makeMove(dust2, player1, ch, ballistics); //reads user input, we need
        //to also pass ballistic incase projectile is shot
        AI->updateAll(); //AIDispatcher; have the AIs move and do what they have to do to(get bomb,
        //defuse, kill enemies, teamkill, be toxic, etc)
        ballistics->updateAll(); //BallisticDispatcher. Handle any ballistics
        dust2->clearScreen(); //Clean up before we re-render
        dust2->renderLevel(); //re-render

        //Check Win Condition here (Time is out or Bomb has been defused etc etc)
        if(dust2->checkRoundStatus() != 0){break;}
    }

    delete map; map = NULL;
    delete dust2; dust2 = NULL;
    delete ballistics; ballistics = NULL; //gotta hide the evidence lol
    delete AI; AI = NULL; //we gotta delete it else it will grow into Skynet, then it's really gg D;
    //Don't delete player1 here, just handle that in the level
    printf("\n gg ez\n");
    endCurses(); //END CURSES
    return 0;
}
```

On the next page you can see an image you can zoom in and see a color coded version of the same code. The main is what you'd expect it to be, concise and made up of object creations and function calls. First we load the map text file into a `CharMap` object (2D char array). Then we initialize `ncurses` library and create our level object and our human player object. The `openNteamSelect` function will also start up the game by asking user what team he wants to join and renders the first frame of the game. Next the `Ballistic Dispatcher` is created using the level as a parameter for the constructor. Finally the AI

Dispatcher is created which takes in both the level and the Ballistic Dispatcher objects. Well references to the objects but you know what I mean. Then we enter the main program loop which will iterate once per tick. In it we have a very important sequence of function calls:

- First we read user input with `readKeyinput`
- Then we parse that input and actually move the human player and/or shoot using the `makeMove` function. Notice this needs the level, the player, the direction, and the ballistics engine to work.
- Next the AI runs with the `updateAll` function. This does any checks it has to and culminates in making its own internal calls to the `makeMove` function so the AI can move and shot.
- Next the ballistic engine runs so it can update ongoing projectiles by 1 grid point as they move towards a wall or a target. Any new projectiles made by the `makeMove` function calls by the AI or main will have been included in the `projList`.
- Now that all the engines ran we `clearScreen` and `renderLevel` which results in creating another frame with all of the updated information (players and bullets have moved)
- Finally we want to `checkRoundStatus` to see if any of the actions that happened in this tick are grounds to end the game such as all CT players having been killed. This function also updates timers such as bomb countdown and round time also checking if either of these indicate the end to a round (e.g. bomb exploded) If any of these result in game over then the function returns a value other than 0 which triggers the break from the main loop.
- At this point since the game is a single round we will go ahead and clean up and terminate main by deallocating memory and calling `endCurses`. We could wrap up this loop in an outer loop that runs for the number of rounds if we wanted to have a multi-round game in which case we would need a *persistent* class that keeps track of stats between rounds such as how many rounds each team has won and so on.

```
int main(int argc, char **argv){
// srand(time(NULL)); //Comment out to always have the same RNG for debugging
CharMap *map = (argc == 2) ? new CharMap(argv[1]) : NULL; //Read in input file
if(map == NULL){printf("Invalid Map File\n"); return 1;} //close if no file given
initCurses(); // Curses Initialisations
//map->print();
Level* dust2 = new Level(map);
player_t* player1 = dust2->openNteamSelect(); //Add Human Player
BallisticDispatcher* ballistics = new BallisticDispatcher(dust2); //handles ballistics for the game. It's cleaner if it is not part of level
//Next we will create the AI Dispatcher that will handle the Artificial Intelligence (Skynet)
AIDispatcher* AI = new AIDispatcher(dust2, ballistics); //This also adds all players (human/bots) to AI so it can manage the pathfinding
//Main Loop
char ch;
while((ch = MovementDispatcher::readkeyinput()) != 'q') { //Super important to not mess up parenthesis here else major fail.
//This next line is the one we'd need to do asynchronous if we wanted to go that route
MovementDispatcher::makeMove(dust2, player1, ch, ballistics); //reads user input, we need to also pass ballistic incase projectile is shot
AI->updateAll(); //AIDispatcher; have the AIs move and do what they have to do to (get bomb, defuse, kill enemies, teamkill, be toxic, etc)
ballistics->updateAll(); //BallisticDispatcher. Handle any ballistics
dust2->clearScreen(); //Clean up before we re-render
dust2->renderLevel(); //re-render

//Check Win Condition here (Time is out or Bomb has been defused etc etc)
if(dust2->checkRoundStatus() != 0){break;}
}

delete map; map = NULL;
delete dust2; dust2 = NULL;
delete ballistics; ballistics = NULL; //gotta hide the evidence lol
delete AI; AI = NULL; //we gotta delete it else it will grow into skynet and then it's really gg 0;
//Don't delete player1 here, just handle that in the level
printf("\ngg ez\n");
endCurses(); //END CURSES
return 0;
}
```

Image of my main color coded. Circa: May 2020.

With this I also conclude the summary of my implementation. It turns out that the explanation is longer than the code, but such is the way of documentation.

I hope that the documentation was not only useful in giving you a good starting point for your own implementation but also helps solve some logical decisions such as how to distribute the program into separate classes and avoid those nasty circular dependency errors. Furthermore I hope this project gives you some insight into game development and the quirks that this sort of programming causes and hopefully handles.

At this point all that is left to say is, take it one function at a time and then one class at a time and test every time you add a new feature to help debugging the issues you will most likely encounter. I would first work on rendering the game using the point and level classes. Then I would add the movement, then the ballistics, and then the AI. This makes it much easier to progressively test as you add features.

Well, it only took 14,100 words and 16 hours to write this document and 30+ to develop and code the project. I could have avoided all of this and just told you to implement Dijkstra and do some trivial thing, but I think this will far better prepare you as a coder and critical problem solver. So hopefully you as a student give me the respect of taking the time to do a good job and start early. Take pride in your work as I did in creating this assignment for you.

# **When you see a bug in CS:GO or Valorant after doing AST12**



# **I built this in my man cave with c++ and ncurses. Why can't they get it right!**

## Closing Notes:

- Comment and Document your source code appropriately and thoroughly
- Make sure you name your program file in accordance with the syllabus stipulations
- Make sure you do all parts of the assignment, and they work!
- Make sure your game and the AI works with any valid map file that students submit or I make!
- Memory leaks to a certain degree will be acceptable. But should be kept to a minimal. This does not mean to just forget to de-allocate everything but that if you have issues fixing some of the tricky leaks, you can let them be and you won't be punished as long as you discuss them in your write up.
- Oh and because you're probably wondering, when I played CS:GO my best rank was LEM but my consistent rank on Average was DMG. This was post-rank changes that happened before I started playing. But that was a long time ago and now I'm probably silver in terms of aim. Such is life

## Extra Credit Opportunity:

Create your own maps based on existing maps for any of the games mentioned so that we can all use them to test our games. Each map that passes my quality control will award you 50 points. You may submit up to 8 of them (400 points total), but each submission must be a map that has not already been done by another student. Special exceptions may be made for custom maps you make but they will be up to my judgement if I consider them to be of good quality.

## Part II:

When completed, create and submit a write-up (PDF Format):

- At least 4 pages long. Include at least 2 photos of your game running on DUST\_2 (on one photo) and then a map of your choice for the second photo. More photos of special situations are welcomed too!
- Name, Assignment, Section
- Summary explaining the implementation of the game. Think of this as doing the README file of your game.
- Discuss the pathfinding algorithm used and its efficiency. How can you improve its space, and time, complexity. If you did not use Dijkstra discuss what you chose and why.
- Discuss how your AI ran and any issues it has and how can you overall improve the AI in the paid DLC.
- A discussion on what was easy, and not so easy and how you overcame any issues you ran into. For examples of this see what I wrote on discord on the bugs I came across.
- If you modified my AI logic above. Explain why and how yours works.
- As mentioned above a certain degree of memory leaks is acceptable. Discuss the leaks you currently have in your submission and where you suspect the issue is coming from and what you could do to try and fix them.
- Now that you have coded game logic, how is it different than other assignments where you coded an algorithm? Is game logic and Algorithm logic the same or are there different sort of styles to them?
- Closing thoughts on assignment, the class as a whole and a quick discussion to the following question:
  - At this point, do you feel prepared for a developer interview and a programmer job/internship?

It's been an honor coding with you this semester. Have a great summer!

*This assignment and any related files or documentation may not be distributed without the explicit consent from Dr. Jorge Fonseca. This includes publishing your solution of the assignment on any public repository or website. All registered trademarks in this assignment are used under the Fair Use Clause provided for Educational purposes in Title 17 U.S.C. Section 107 of the US Copyright Law. All derivatives of this work may not be distributed without the explicit written consent from Dr. Jorge Fonseca.*