

C++ Day 3

Introduce Object- Oriented Programming

Object-Oriented Programming

Using OOP, we can create our own data type.

For example, if we need to keep track of the statistics of all characters in our game:

Each character will need a name, health (HP), attack damage (AD), attack resistance (AR), and a list of items carried by the character.

We can create a data type called "character", which maintains all these statistics of a single character.



Object-Oriented Programming

Class

We combine related variables and functions into a unit to describe a real-world concept.

"Character" is a class, "Player" is a class.

Object

We can create many instances of a class, each of these instances is called an object.

A character called Bob is an object (a character instance), a different character called Alice is also an object (another character instance).

Object-Oriented Programming

Property

Properties are variables defined in a class.

name, health (HP), attack_damage (AD), attack_resistance (AR), list of items are properties of a character.

Method

Methods are functions defined in a class.

Respawn() is a method of a character. All characters can respawn, but they may respawn with different HP, depending on their health.

```
10 class Character {
11 public:
12     Character(string name, int hp, int ad, int ar){
13         // some code for initialisation...
14     }
15
16     string name;
17     int hp, ad, ar;
18     vector<Item> items;
19
20     void respawn(){
21         // some code for respawn...
22         cout << "Respawn with: " << hp;
23     }
24 };
25
26 int main() {
27     Character bob("bob", 2000, 180, 100);
28     Character alice("alice", 1000, 300, 200);
29
30     bob.respawn();
31     alice.respawn();
32
33     return 0;
34 }
```

Daily Code Jumpstart Choreography

	Mon	Tues	Wed	Thurs	Fri
9am-10am	---	Coaching aims	Daily Aims and Objectives	Daily Aims and Objectives	---
10am-13oo	---	Self-study Time	Self-study Time	Self-study Time	---
Break	---	Social Lunch	Social Lunch	Social Lunch	---
14oo-16oo	---	Self-study Time	Self-study Time	Self-study Time	---
18oo-19oo	---	QandA with Coach	QandA with Coach	QandA with Coach	---

Day 3 Tasks

- Task 1 – Two Videos on Object-Oriented Programming (approx. 40 mins)
- Task 2 – Upgrade the Tic-tac-toe (approx. 50 mins)

Task 1

1. Object-Oriented Programming

Task 1.1 - Introduction to OOP (<10 mins)

Watch this [Introduction to OOP](#) video about the four main principles in OOP

Task 1.2 - OOP in Practice (~30 mins)

Watch and follow this [Practical tutorial on OOP in C++](#) (only 3:13:27 to 3:41:42 [Chapter Classes & Objects, Constructor Functions, and Object Functions])

Task 2

2. Better Tic-Tac-Toe

Recap:

Yesterday we made our basic tic-tac-toe game:

- we used a 2D array to represent the grid
- we also designed a program loop to model the interaction

```
9  int grid[y_dim][x_dim]; // grid: 0 for empty; 1 for player x; 2 for player o;
10 int marker; // marker: 1 for player x, 2 for player o;
11
12 void initialiseGrid();
13 void showGrid();
14 bool checkInput(int x, int y);
15
16 int main() {
17
18     int x,y;
19     initialiseGrid();
20     showGrid();
21
22     for (int i = 0; i < x_dim * y_dim; i++){
23
24         marker = i % 2 + 1;
25
26         while (true) {
27             if (marker == 1){
28                 cout << "Player x \n";
29             } else {
30                 cout << "Player o \n";
31             }
32             cout << "enter row: ";
33             cin >> y;
34             cout << "enter column: ";
35             cin >> x;
36
37             if (checkInput(x, y)){
38                 break;
39             }
40         }
41         grid[y-1][x-1] = marker;
42         showGrid();
43     }
44 }
```


Task 2

2. Better Tic-Tac-Toe

Task 2 - Better Tic-Tac-Toe (approx. 50 min):

Today we're going to upgrade our game to automatically check which player is the winner.

As our program grows, we may end up with a bunch of variables and functions all over the place.

Therefore, we're going to use OOP to organise our code.

More instructions on following slides ->

```
player x:
enter row: 2
enter column: 2
x [ ][ ]
o x [ ]
[ ][ ][ ]

player o:
enter row: 3
enter column: 2
x [ ][ ]
o x [ ]
[ ] o [ ]

player x:
enter row: 3
enter column: 3
x [ ][ ]
o x [ ]
[ ] o x

player x win
```

Task 2

2. Better Tic-Tac-Toe

Step 1 – Define System Components

We'll split the system in a Grid object and a Player object.

```
class Grid{};
```

The grid object handles everything happens on the grid, acting like a referee (i.e. add markers, initialise and print the grid, decide a winner, check if a cell is taken)

```
class Player{};
```

The player object represent a player, maintaining a player's index and markers, and handle inputs.

Task 2

2. Better Tic-Tac-Toe

```
class Grid{};
```

Properties:

```
int x_dim;  
int y_dim;  
vector<vector<int> >grid;
```

x_dim and y_dim represent the dimension of the grid, in a 3 x 3 grid they'll all be 3

Methods:

```
Grid(int x_dim, int y_dim);  
void initialiseGrid(int x_dim, int y_dim);  
void showGrid();  
bool isGameOver();  
bool checkInput(int x, int y);  
void placeMarker(int x, int y, int marker);  
  
int checkRowCrossed();  
int checkColumnCrossed();  
int checkDiagonalCrossed();
```

the constructor function that is going to run every time we define a new Grid object

checking every row, column and diagonal to see if there is a winner, will be called by the `isGameOver()` function (since the game will end if a winner shows up)


Task 2

2. Better Tic-Tac-Toe

```
class Player{};
```

Properties:

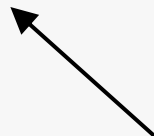
```
int index;  
char marker_char;
```



character "X" or "O"

Methods:

```
Player(int player_index);  
void playersMove(Grid &grid);
```



there's a "&" in front of the grid, it means we are passing a reference of the vector, instead of passing a copy of it, so changes made in this function is reflected in main()

Task 2

2. Better Tic-Tac-Toe

Step 2 – Check Winner

Right after a player makes a move, we check all rows, columns and diagonals.

When checking rows, we iterate through all rows and check if there's a row with three non-zero marks that match.

```
95     int checkRowCrossed(){
96         for (int i = 0; i < y_dim; i++){
97             int n = 0;
98             while (grid[i][n] == grid[i][n+1] and grid[i][n] != 0){
99                 n += 1;
100                 if (n == x_dim - 1){
101                     // If we find two pairs of matching marks in a row, return with the number at that place.
102                     return grid[i][n];
103                 }
104             }
105         }
106         // If we checked all rows and found nothing, return 0
107         return 0;
108     }
```

We use similar approach to check columns in `int checkColumnCrossed();` and check diagonals in `int checkDiagonalCrossed();`

Task 2

2. Better Tic-Tac-Toe

Step 3 – Is Game Continue?

To decide whether the game is continuing after each move, we first **run the three checking functions** we defined in the last step, then **check if there are empty spaces** in the grid.

The game is not over if no winner shows up and there are still empty spaces.

```
64     bool isGameOver(){
65
66         // we are checking on every rows, columns and diagonals to see if there is a winner
67         // these three functions return 0 if there's no winner, return 1 or 2 indicating player 1 or 2 has win the game
68         int row = checkRowCrossed();
69         int col = checkColumnCrossed();
70         int dia = checkDiagonalCrossed();
71
72         if (row==1 or col==1 or dia==1){
73             cout << "player x win\n";
74             return true;
75         } else if (row==2 or col==2 or dia==2){
76             cout << "player o win\n";
77             return true;
78         }
79
80         // next, we check if the grid has no empty space, in that case the game ends with a draw
81         for (int y = 0; y < y_dim; y++){
82             for (int x = 0; x < x_dim; x++){
83                 if (grid[y][x] == 0){
84                     // if we spot an empty space, return false so that the game is not over
85                     return false;
86                 }
87             }
88         }
89         cout << "draw\n";
90         return true;
91     }
```

Task 2

2. Better Tic-Tac-Toe

Step 4 – Put Everything Together

It's a good practice to keep the main function minimal, and only contain high-level processes, so that our program is more maintainable.

[Link to the full code](#)

```
201 int main() {  
202  
203     Grid grid(3, 3);  
204     Player player1(1);  
205     Player player2(2);  
206  
207     int round = 0;  
208     grid.showGrid();  
209  
210     while(!grid.isGameOver()){  
211         if (round%2 == 0){  
212             player1.playersMove(grid);  
213         } else{  
214             player2.playersMove(grid);  
215         }  
216         grid.showGrid();  
217         round += 1;  
218     }  
219 }
```

Initialisation

Main game loop

Day 3 resources

[CreativeApplications.Net](#): a community of art, media and technology

[GitHub Quickstart](#)

[Missing Semester](#): a short course help you to gets familiar with terms like terminal, Linux command-line, git, version controls...

[Map of Computer Science](#): a 10mins video explaining main subjects in computer science

Outlook: C++ at CCI

[openFrameworks](#): a tool for creative coding (e.g. interactive moving images, generative arts / sounds, visualisation...)

[Raspberry Pi](#): run openFrameworks in embedded systems for installations, synthesisers.

[JUCE](#): a tool for making music app / plug-ins / virtual instruments

[Unity](#): game engine

[Unreal Engine](#): game engine



Concluding Week 3 Survey

<https://artslondon.padlet.org/hbrueggemann/j2yr3zfwkap4v4rq>

The password is **Jumpstart**.

Final Code Jumpstart Survey

[Link to survey]

Thank you for joining 😊
Catch you at Welcome Week!

 @ual_cci

 @ual_cci

arts.ac.uk/cci