CT4019 PROGRAMMING AND MATHEMATICS

Loops

Attendance PIN

xxxx


TAKELINGS
HOUSE PARTY
GANG BEASTS
WORMS 3D
RETRO GAMES
Open to everyone!
LAN party
15:00
WW001
DONUTS
Wednesday
9th October
STAR WARS
BATTLEFRONT II

Last week's reading…

Q) What is the 'Dangling Else' problem?

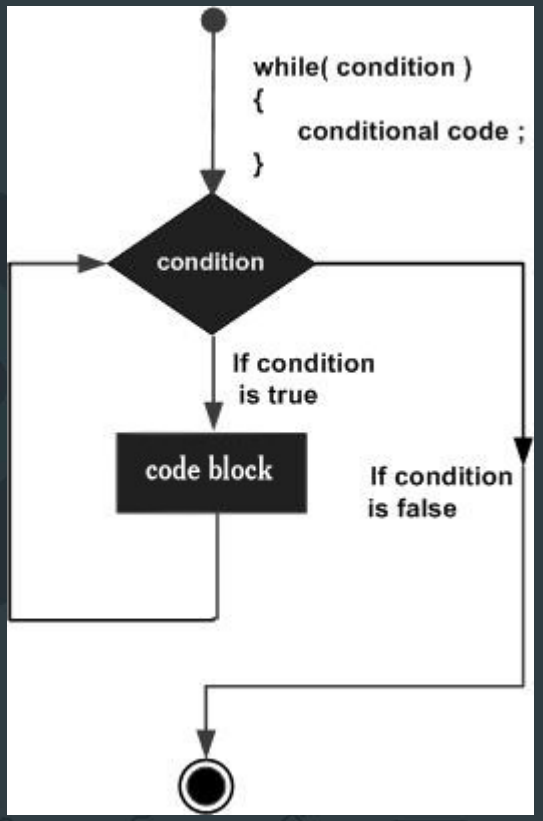
– if (a) if (b) s; else s2;

Q) What is the 'conditional operator'?

– x == y ? a : b

Loops

- Loops are among the most basic and power programming concept.
- A computer game is essentially just a loop (or a series of loops).
- Loops are a combination of conditions and logic execution.
- And there are a few different types.



Types of Loop

Count Loops

- A loop that repeats a given number of times.
- Possibly based on user input.
- Possibly based on the size of a dataset.
- Potentially it has a set number of iterations.

Sentinel Loops

- A ‘sentinel’ is essentially a flag.
- This flag tells the loop when to stop iterating.
- A particular value or flag acts as our ‘sentinel’.
- Reading valid input is one example of this.

Conditional Loops

- Similar to sentinel loops but they terminate as the result of a calculation.
- Kind of a combination between an **if statement** and loop.
- A game based example: Looping over AI firing code until we are out of ammo.

The “**while**” loop

- The simplest form of loop is the “**while**” loop
- Similar to an “if” statement, this loop will execute all logic within the braces while the expression evaluated to **true**
- It will repeat this logic again and again until the expression evaluates to **false**
- The while loop will check the expression in the brackets **before** each iteration
- In this example the variable ‘i’ starts at zero, the while loop will iterate through printing out the value of ‘i’ and incrementing it by one
- Once the value of ‘i’ is equal to 10 the loop will exit
- Be wary of **infinite loops**!

```
while ( expression )
{
    // logic
}
```

```
while (i < 10)
{
    // Print the current iteration
    std::cout << "Current iteration: " << i;

    // Increments i by 1
    i++;

    // Pause
    system("pause");
}
```

Task #1 – while(user == gullible)

1. Using a *while* loop write a program that continues to ask the user to enter any number **other than** 5 until the user enters the number 5.

2. When the number 5 is entered tell the user: *"Hey you weren't supposed to enter 5!"* and exit the program.

3. If the user has **not** entered the number 5 after 10 iterations tell the user : *"Wow, you're more patient than I am, you win."* then exit the program.

The “**do while**” loop

- The “do while” loop is almost identical
- However it checks the expression in the brackets **after** each iteration
- This means using a **do while** will always guarantee the logic in the braces runs **at least once**
- This example will perform exactly the same as the previous example except the expression check is done after the logic
- Useful for final condition checking

```
do
{
    // logic
} while ( expression )
```

```
do
{
    // Print the current iteration
    std::cout << "Current iteration: " << i;

    // Increments i by 1
    i++;

    // Pause
    system("pause");

} while (i < 10);
```

Task #2 – do User Input check

1. Take the code from one of your previous week's tasks that asks for user input.
2. Add a do-while loop that will continue to ask the user for valid input when they enter something incorrectly.
3. The program should continue to loop until valid input is entered and then complete.

Example Output

Please input a whole number: a

The input was in invalid.

Please input a whole number: cheese

The input was in invalid.

Please input a whole number: -5.5

The input was in invalid.

The “**for**” loop

- The most common type of loop you will come across is the **“for”** loop
- A **for** loop requires an initialisation, usually where a counter variable is declared and set to an initial value
- A condition is checked at the start of each iteration, as long as the condition is true the loop will run
- And an increase that should progress the loop counter
- Each iteration the loop will execute the logic within its braces
- This example shows a loop that will iterate 10 times, each time printing out the current iteration number and then pausing for user input.

```
for( initialisation; condition; increase )
{
    // logic
}
```

```
for( int i = 0; i < 10; i++ )
{
    std::cout << "Current iteration: " << i;

    system( "pause" );
}
```

Task #3 – for Natural Numbers

1. Ask a user to enter a whole number 1-9.
2. Use a for loop to write a program that will print out all whole numbers up to and including the number entered.
3. Additionally print out the sum of those whole numbers.
4. Make sure you account for any invalid input.

Example Output

Please input a whole number: 7

The whole numbers are: 1 2 3 4 5 6 7

The sum is : 28

Nested Loops

- So we know how to make a basic loop.
- However solving more complex problems we will often need to use a series of loops.
- For instance, to iterate over rows and columns of data.
- Why is this important to game programming?

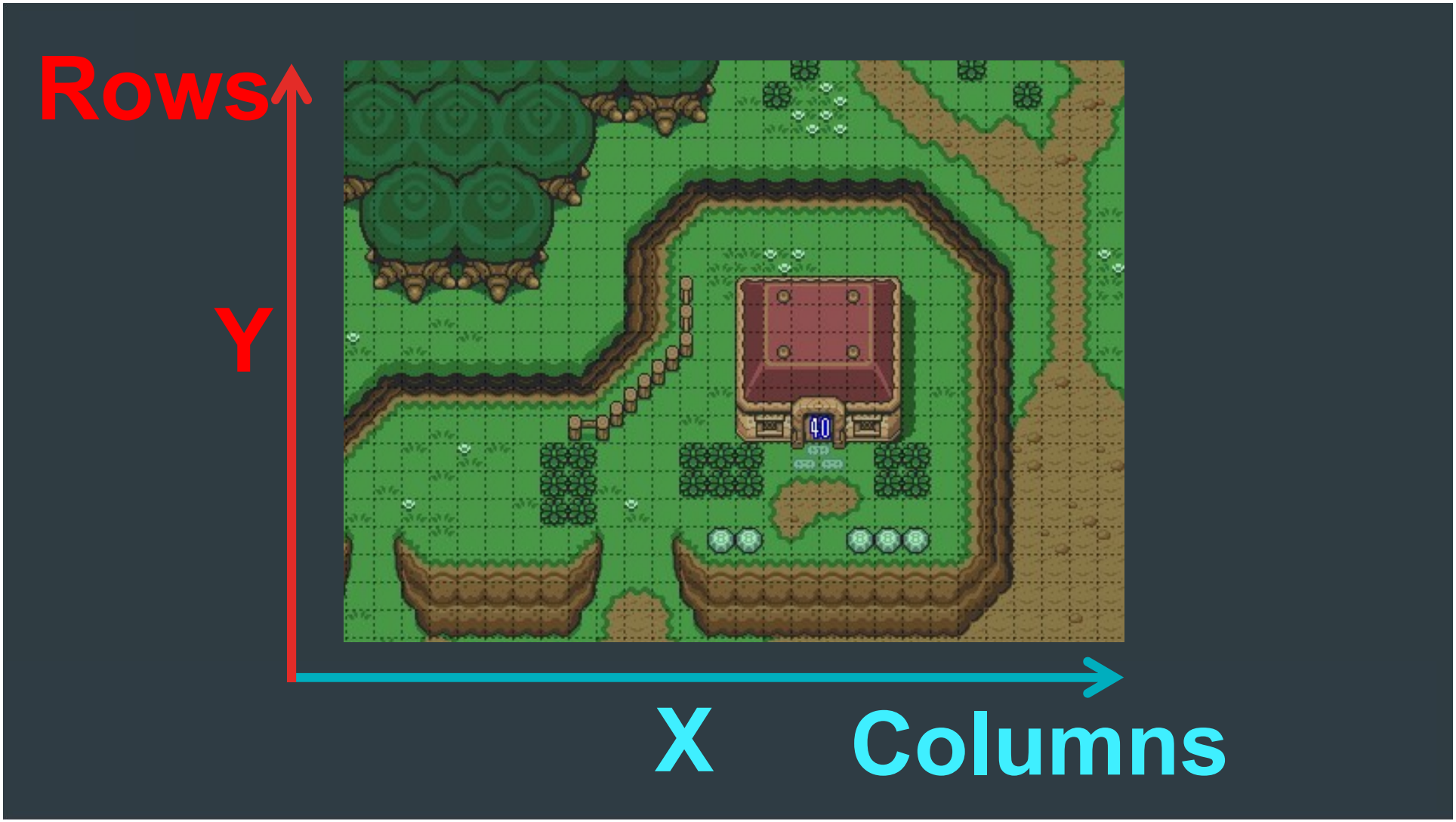
```
for(num1 = 0; num1 <= 9; num1++)
{
    cout << num1 << endl;
}
```

```
for(num2 = 0; num2<=9; num2++)
{
    for(num1=0; num1<=9; num1++)
    {
        cout<< num2 <<"  "<< num1<< endl;
    }
}
```

outer

inner

	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]


Rows
Y
X
Columns

Task #4

1. Ask a user to enter a whole number 1-9.
2. Use a nested loop to draw a diamond of asterisks of that size entered by the user.
3. For instance, if the user enters 5, the diamond will look as it does here ---->

Most common “Gotchas”

- Creating an **infinite loop** whereby the condition for a loop will never be met so the program hands indefinitely
- Forgetting the logical operator **“!”** inverts any Boolean value
- Don’t forget as programmers we start counting from **0** not from 1!
- Not incrementing the counter of a loop for every iteration

Recursion

- A fancy term for when a function calls itself.
- A function is a collection of code statements.
- Our entry point, main, is a function we have to implement.
- We can write our own functions.
- And then *'call'* that function in main.

```
int main()
{
    int doubled = doubleNumber(5);

    return 0;
}


int doubleNumber(int x)
{
    int result = x * 2;

    return result;
}
```

Recursion

- Solving a problem recursively we call a function.
- Which calls itself.
- Here we have a recursive function that takes in an integer.
- It then checks if the integer taken in is **NOT** zero.
- If it's not we add the current integer to the result return from another call of this function minus.

```
int main()
{
    int sum = addNumbers(5);

    return 0;
}

int addNumbers(int n)
{
    if(n != 0)
    {
       return n + addNumbers(n-1);
    }
    else
    {
       return n;
    }
}
```


Any questions?

Final Task – Text Adventure Game

Using the techniques from the last two sessions write a Text Adventure Game based on a "**Haunted House**" setting which includes the following:

- Give the player the ability to go North, East, South or West by entering a number from 1 to 4
- Have one locked room for which the player must find and pickup a key before they can enter
- Give the player a "Hitpoints" value, reduce this value when the player enters a room with a trap in it
- If the player's hitpoints reach zero, the game is over and they must start again
- If the player dies, ask if they would like to play again or quit
- If the player gets to the final room and wins the game, ask if they would like to play again or quit.

Example output

"You are in a room with a table in the centre, there is a key on the table. There is a door way on each wall.

1. Go North
2. Go East
3. Go South
4. Go West
5. Pickup key"

"You go North only to trip and fall! -25 HP"

"You died!"

"You try to open the door but the door is locked, perhaps you need a key?"