Intro to CS #2

By Sadek Mohammed

1. Run the program.

The program is simple; however, there are some steps that should be followed.

1. click *create new project*
2. choose c++
3. use *console application*
4. choose a name for the project
5. Do not edit any of the file extension stuff.
6. Remember to add a path to the project in order to avoid empty projects.

2. Casting

2.1. Definition & Syntax

Changing of a datatype into another one for a specific reason.

It is written by enclosing the target datatype with parenthesis.

```
cout << (char) 73 << endl;
```

2.2. Uses

2.2.1. The decimal division syndrome.

In the last lecture, the basic datatypes and their operations were discussed. One of the operations was the division. For example, imagine dividing 5 by 2.

```
cout << 5 / 2 << endl; // The expected answer is 2.5
```

Since the expected answer is 2.5, it was surprising to see a "2" in the output console.

2.2.1.1. Explanation:

Since both 5 and 2 are integers by default, the compiler assign the datatype "int" to the resultant of their operations. Since integers cannot hold decimal points, the compiler ignores the ".5" part of the number and outputs 2.

2.2.1.2. How to fix?

If we could tell the compiler that we need the output in the decimal point, the problem would be solved. Although it may look weird to do so, the c++ language enables us to change the type of several datatypes into other ones. This is called spreading.

```
cout << (float)(5 / 2) << endl; // The output is 2.5 as expected
```

Note: Casting to a double could have been a possible solution.

2.2.2. Avoiding overflows

In the normal case, that code will overflow and give random results.

```
    int x = pow(2, 30); // The pow function is in the
    cout << x << endl;
    cout << x * x << endl;
```

To solve that problem, just cast into a long long.

```
cout << (long long)(x * x) << endl;
```

pow is a function in the `<cmath>` and `<math.h>` Although adding `#include<cmath>` into our code may look like a potential solution, it would be both unbearable and time-consuming to include a library for each function we use. That's why competitive programmers use the following include only, where it includes all other includes.

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    // The new configuration for coding.
}
```

2.2.3. Chars

Remember the code `cout << (char)73 << endl` from the last lecture. That line outputs a capital "I" in the console.

3. Array

3.1. What is an array?

During the last lectures, variables were used like this:

```cpp
int studentOneMark = 25; // An expression that initializes a variable
```

Imagine that you have 10 students! It would be impossible to keep track of ten different variables, so arrays were invented. The code of an array is like this.

```cpp
int studentsMarks[10] = {25, 24, 23, 22, 21, 18, 22, 16, 25, 17};
```

3.2. Array Operations

In order to access the elements of an array, indices are used. C++ and most programming languages are zero-based. That means that if an array has the length n, it has indices from zero to $n - 1$.

That implies that if we were talking about the element in the second index, we are talking about the third element. If we were talking about the element in the zeroth index, we are talking about the first element and so on.

Arrays cannot be multiplied, divided, added, subtracted, powered, appended to other arrays, or even append other elements to it. The only available operations are accessing certain indices or modify elements at certain indices.

See the following codes that talk about the above array

```cpp
cout << studentMarks[3] << endl; // Output is 22
cout << studentMarks[2] - studentMarks[9] << endl; // Output is 6
cout << studentMarks[10] << endl; // Invalid: There is no such index.
```

4. Loops

Imagine that you have the following piece of code.

```cpp
cout << "Hello" << endl;
cout << "Hello" << endl;
cout << "Hello" << endl;
// Those lines of code output "Hello" three times, each in a line.
```

Imagine that you need to save that effort by writing an easier block of code. Here comes the advantage of loops, which is repetition.

```
//   initialize variable; condition; loop step
for (int i = 0; i < 3; i++)
{
    // body of the loop: orders.
    cout << "Hello" << endl;
}
```

To understand loops well, it is mandatory to discuss some concepts such as: scope & Boolean expression.

4.1. Related Topics

4.1.1. Scope

To use a variable, it is necessary to declare it, where for each line of code, there is a scope(range of lines of code) that the variable could be manipulated within.

4.1.1.1. Examples:

If a variable is declared out of a function, it is visible in all of the program's functions that are below it.

If a variable is declared inside a certain function, it is visible in all of the function's lines that are below it.

If a variable is declared inside a certain code block, such as a loop or if condition (to be explained), it is visible in all of the block's lines of code away from it.

4.1.2. Boolean Expressions

Trough the discussion of data types, Booleans were explained. Booleans are the datatypes that denote true or false. Other expressions are evaluated by the compiler and assigned Boolean values.

- x == 5 (Equality operator; checks whether two quantities are equal or not)
- x != 5 (Inequality operator; checks whether two quantities aren't equal or not)
- x < 5 (Less than operator; checks whether quantity is less than another one or not)
- x > 5 (More than operator; checks whether quantity is more than another one or not)
- x <= 5 (Less than or equal operator; checks whether quantity is less than or equal another one or not)

- x >= 5 (More than or equal operator; checks whether quantity is more than or equal another one or not)

4.2. Follow Loops:

During a loop, the compiler initializes the first value in the expression. After that, it checks the condition, and if that expression yielded true, the statements in the body of the loop are executed. After that, increment operator is used to cause a change in the value of i. After changing the i variable, the compiler checks the condition again. If that condition is true, the loop is continued and so on.

- [ ] The initialized variable is appropriate and does not interfere with any other variables of a higher scope.
- [ ] The step could reach the condition. For example, if i is initialized to 5 and the condition is to stop when i reaches 8. That's why making the loop step to decrement i is meaningless as i won't reach 5 by subtracting ones.