SDL2 Course Day-1

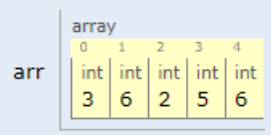
Cont. Intro to C++

C is painful to use (Suhas thinks it's fun). So, let's see how C++ makes our life better :)

Arrays:

How arrays are represented in memory:

```
int arr[5] = {3,6,2,5,6};
```



Range based for-loop to access elements of arr:

```
int arr[5] = {3, 8, 2, 5, 1};

for (auto i: arr) {
    std::cout << i << " ";
}
```

Range based for-loop with reference to change the elements of the arr:

```
int arr[5] = {3, 8, 2, 5, 1};

for (auto &i: arr) {
    i = i + 1;
}
```

Unless you absolutely need to use arrays, use vectors, they're way better and easier to use.

Structures:

Arrays, vectors and stuff only contain elements of the same type. Not able to model real-life situations.

Custom User-defined data type needed = > structure

```
struct my_struct{
    int x_pos;
    int y_pos;
    float dist;
};
```

creates blueprint / standard; here x_pos, y_pos and dist are called "members" of "my_struct";

```
my_struct a;
```

"instance" is defined by using it like a datatype.

Accessing elements of structure: (use “dot”)

```
a.x_pos = 3;
a.y_pos = 4;
a.dist = 4.1;
```

Pointers to structures:

Anything can have a pointer to it;

Syntax for pointer to int:

```
int a;
int *p;
p = &a;
```

creating a pointer “p” pointing to variable “a” of type “int”;

```
*p
```

accessing value of “a” using “p”;

Syntax for pointer to struct: (replace int with your structure):

```
my_struct a;
my_struct *p;
p = &a;
```

creating a pointer “p” pointing to variable “a” of type “my_struct”;

```
*p
```

accessing value of “a” using “p”;

Now a’s members are accessed using “a. x_pos”

Since “ *p ” is the same as “a” hence “(*p). x_pos” will work;

```
a.x_pos
(*p).x_pos
```

both of these mean the same thing

This is kinda painful syntax so GNU peeps made a symbol for this:

```
a.x_pos
(*p).x_pos
p->x_pos
```

so remember, whenever you’re not using a pointer, use (.) ; whenever using a pointer to a structure, use (->) ;

Dynamic Memory:

In C/C++ there exists two types of memory: Stack memory and Heap memory.

Stack memory / static memory, is memory allocated at compile time (not executed yet)

Heap memory / dynamic memory, is memory allocated at run time (during execution)

There are many reasons why we use heap memory, one of the main reasons is to follow the principle of use resources only when needed.

Syntax for creating dynamic memory using pointers:

```
int *p = new int;
*p = 4;
```

The above two statements can be combined into one by saying:

```
int *p = new int(4);
```

Syntax for deleting dynamic memory:

```
delete p;
p = nullptr;
```

It is always good practice to assign a pointer to “nullptr” after deleting it. The default initial value of an integer which everyone uses is 0. In the same way “nullptr” is the default initial value which people use for a pointer. (“nullptr” basically means empty / pointing to nothing).

NEVER allocate memory if its already allocated before and not deleted => leads to memory leak.

Classes:

Classes are the basis of Object Oriented Programming (OOP). Classes model real world objects which have properties (member variables) and behaviour (member functions). Classes act as user defined data types. The user decides the type of attributes and functions according to his/her requirement.

When you create a variable whose data type is a class, that variable is called as an “Instance” or an “object” of that class. The class acts as a blueprint and provides a standard which is followed by the objects.

The syntax for classes and structures are the same except you have to replace the “struct” keyword with “class” and also put a “public:” inside your class. Search online and learn about “public”, “private” and “protected” if you want, we will not be using anything apart from “public” anywhere.

An example:

A car is a real world entity that we can model as a class. A car has properties like model, make and speed. It has behaviour which is to accelerate.

The above model can be described in code as the following:

```
class Car {
public:
    int model;
    string make;
    int speed;

    void accelerate(int sp_inc) {
        speed += sp_inc;
    }
};
```

Whenever an instance of a class is created a “constructor” is called, and when an instance goes out of scope (not accessible anymore), a “destructor” is **automatically** called. (constructor creates, destructor destroys).

Constructors are of 2 types **non-parameterized** (doesn’t have parameters) and **parameterized** (has parameters).

1.) Non – Parameterized:

```
Car(){
    model = 0;
    make = "audi";
    cout << "Car's constructor" << endl;
}
```

creation of instance: `Car c1;` or `Car c1();`

2.) Parameterized:

```
Car(int x, string y){
    model = x;
    make = y;
    cout << "Car's constructor" << endl;
}
```

creation of instance: `Car c1(4, "audi");`

The problem with parameterized constructor is when you have same names for the parameters and the member variables.

```
Car(int model, string make){
    model = model;
    make = make;
    cout << "Car's constructor" << endl;
}
```

This confuses the compiler as it doesn't know which model or make we are talking about.

Using **this->** keyword to fix the problem:

```
Car(int model, string make){
    this->model = model;
    this->make = make;
    cout << "Car's constructor" << endl;
}
```

Using **initializer list** to fix the problem:

```
Car(int model, string make) : make(make), model(model){
    cout << "Car's constructor" << endl;
}
```

Both of these are just different syntax to solve the same problem so anything works. We prefer the 2nd.

A destructor is used to destroy an instance when it is no longer in use. If not defined, they are automatically implemented and for now we won't need to define or modify them unless we use dynamic memory.

```
~Car(){
    cout << "Car's destructor" << endl;
}
```

Creating header files:

```
#include <iostream>

int add(int a, int b) {
    return a + b;
}


int main() {
    std::cout << add(4, 3);

    return 0;
}
```

Lets say we have a function add which has two integer parameters and returns the sum of them.

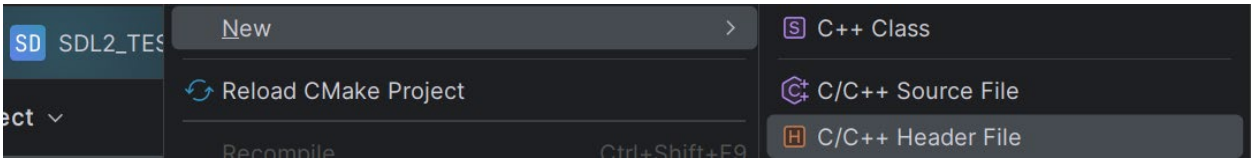
This function is currently defined in the main.cpp file. In the future when we define multiple functions and classes, we cannot put all of these functions in a single file. So we need a way to split code across multiple files.

The first step is to declare the function instead of defining it and put the definition of the function anywhere else;

```cpp
#include <iostream>

int add(int a, int b);  declaration

int main() {
    std::cout << add(4, 3);

    return 0;
}

int add(int a, int b) {
    return a + b;
}
```
definition

After we split the function into declaration and definition, we put the declaration in a **.h file** (called as header files), and we put the definition in another .cpp file.

Clion gives the functionality of creating header files from the menu which we will use.



When you create a new header file, it automatically creates the below code:

```cpp
#ifndef SDL2_TEST_HEADER_FILE_TEST_H
#define SDL2_TEST_HEADER_FILE_TEST_H

#endif //SDL2_TEST_HEADER_FILE_TEST_H
```

DO NOT DELETE THESE LINES, instead type your declarations in between these lines like so:

```cpp
#ifndef SDL2_TEST_HEADER_FILE_TEST_H
#define SDL2_TEST_HEADER_FILE_TEST_H

int add(int a, int b);

#endif //SDL2_TEST_HEADER_FILE_TEST_H
```

In your main file now at the top we will #include the header file we have created and remove the function declaration:

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

int main() {
    std::cout << add(4, 3);

    return 0;
}

int add(int a, int b) {
    return a + b;
}
```

Now the next step is to put the function definition in another .cpp file. Create another .cpp file (it is good programming etiquette to name the .cpp file the same as the header file its defining).

In this new .cpp file we will #include our header file and then define our function:

```
#include "header_file_Test.h"

int add(int a, int b) {
    return a + b;
}
```

Now in our main file we can remove the function definition.

The last step is to now tell our compiler that we have created 2 new files and these files work together with our main.cpp file. We do this by adding the file names to our **CMakeLists.txt** file:

Before:

```
add_executable(SDL2_TEST main.cpp)
```

After:

```
add_executable(SDL2_TEST main.cpp header_file_Test.cpp header_file_Test.h)
```

After adding this we get an option to **reload cmake changes** on the top right:

 click on this button and you're done.

Finally we have the following files:

main.cpp:

```cpp
#include <iostream>
#include "header_file_Test.h"

int main() {
    std::cout << add(4, 3);

    return 0;
}
```

header_file_Test.h:

```cpp
#ifndef SDL2_TEST_HEADER_FILE_TEST_H
#define SDL2_TEST_HEADER_FILE_TEST_H

int add(int a, int b);

#endif //SDL2_TEST_HEADER_FILE_TEST_H
```

header_file_Test.cpp:

```cpp
#include "header_file_Test.h"

int add(int a, int b) {
    return a + b;
}
```