Introductory Robot Programming

ENPM809Y

Lecture 03 – User-defined Types and Data Structures

Zeid Kootbally
zeidk@umd.edu

Fall 2020

Overview I

Overview II

Overview III

User-defined Data Types

- ReadingMaterial01 summarizes C++ fundamental data types, e.g., `int`, `double`, `char`.
- Many programming languages, including C++, allow users to create their own data types (user-defined data types) according to their needs.
- User-defined data types provide more flexibility and more complexity than fundamental data types.
- In this lecture we will see some of these user-defined data types.
- Create ➤ CLion:Project:Lecture03 with the C++17 compiler.

Enumeration

- An enumerated type (also called an enumeration) is a data type where every possible value is defined as a symbolic constant (called an enumerator).
- Enumerations are defined via the `enum` keyword.
- According to Google C++ Style Guide, enumerators should be named like constants and not like macros.

```
//--define a new enumeration named ColorRedShade
enum ColorRedShade{
    kBurgundy,
    kCrimsonRed,
    kRaspberry,//--trailing comma on the last enumerator is possible since C++11
}; //--the enum itself must end with a semicolon
```

- Note: Each enumerator is separated by a comma, and the entire enumeration is ended with a semicolon.

Enumeration | enum Variable

- Defining an enumeration (or any user-defined data type) does not allocate any memory.
- When a variable of the `enum` type is created, memory is allocated for that variable at that time.

```
//--define a few variables of enumerated type Color
ColorRedShade house{kBurgundy};
ColorRedShade apple{kCrimsonRed};
ColorRedShade car{kRaspberry};
```

Enumeration | enum Scope

- Enumerators are placed into the same namespace as the `enum`.
- An enumerator name cannot be used in multiple enumerations within the same namespace.

```
enum ColorRedShade{
    kBurgundy,
    kCrimsonRed,
    kRaspberry,
};

enum Fruit{
    kBanana,
    kApple,
    kRaspberry, //--Error
};
```

Enumeration | Enumerator Values

- Each enumerator is automatically assigned an integer value based on its position in the enumeration list.
- By default, the first enumerator is assigned the integer value 0, and each subsequent enumerator has a value one greater than the previous enumerator.

```
enum ColorRedShade{
    kBurgundy,      //-- 0
    kCrimsonRed,    //-- 1
    kRaspberry,     //-- 2
};

int main(){
    ColorRedShade apple{kCrimsonRed};
    std::cout << apple << std::endl; //-- 1
}
```

- *Best practice*: Do not assign specific values to your enumerators and do not assign the same value to more than one enumerator in the same `enum`.

Enumeration | Evaluation

- Because enumerated values evaluate to integers, they can be assigned to integer variables.
- This means they can also be output as integers.

```
enum ColorRedShade{
    kBurgundy,      //-- 0
    kCrimsonRed,    //-- 1
    kRaspberry,     //-- 2
};

int main(){
    int apple = kCrimsonRed;
    std::cout << apple << std::endl; //-- 1
}
```

Enumeration | Usefulness

- What is `enum` useful for?
- `enum` types are incredibly useful for code readability purposes when you need to represent a specific and predefined set of states.
- For example, old functions sometimes return integers to the caller to represent error codes when something went wrong inside the function. Typically, small negative numbers are used to represent different possible error codes.

```
int ReadFileContents(){
    if (!OpenFile())
        return -1;
    if (!ReadFile())
        return -2;
    if (!ParseFile())
        return -3;

    return 0; //--Success
}
```

Enumeration | Usefulness

- The previous code snippet makes is hard to read and these magic numbers are not very descriptive.

```
enum FileOperation{
    //--No need for specific values for our enumerators.
    kSuccess,
    kErrorOpeningFile,
    kErrorReadingFile,
    kErrorParsingFile
};

FileOperation ReadFileContents(){
    if (!OpenFile())
        return kErrorOpeningFile;
    if (!ReadFile())
        return kErrorReadingFile;
    if (!Parsefile())
        return kErrorParsingFile;

    return kSuccess;
}
```

Enumeration | Usefulness

```
enum ItemType{
    kSword,
    kTorch,
    kPotion
};

std::string GetItemName(ItemType itemType){
    if (itemType == kSword)
        return "Sword";
    if (itemType == kTorch)
        return "Torch";
    if (itemType == kPotion)
        return "Potion";
    return "???";
}

int main(){
    ItemType itemType{kTorch};
    std::cout << "Link is carrying a " << GetItemName(itemType) << '\n';
}
```

Structure

- There are many instances in programming where we need more than one variable in order to represent an object.
- For example, to represent a person, you might want to store data related to a person, such as `name`, `birthday`, `height`, `weight`, or any other number of characteristics about a person.

```
std::string name;
int birth_year;
int birth_month;
int birth_day;
double height;
double weight;
```

- However, you now have 6 independent variables that are not grouped in any way.
- If you wanted to store information about someone else, you would have to declare 6 more variables for each additional person (with different identifiers).

Structure

- Fortunately, C++allows us to create our own user-defined aggregate data types.
- An aggregate data type is a data type that groups multiple individual variables together.
- One of the simplest aggregate data types is a structure, declared with the keyword `struct`.
- A `struct` allows us to group variables of mixed data types together into a single unit.

Structure

- Because `struct`s are user-defined, we first have to tell the compiler what our `struct` looks like before we can begin using it.

```
struct [identifier] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

- The `identifier` is optional. If you do not provide an `identifier` you will create an anonymous `struct`.
- Each member has a data type (fundamental or user-defined).
- At the end of the `struct` definition, before the final semicolon, you can specify one or more structure variables. This part is also optional.

Structure

```
struct Person {
    std::string name;
    int birth_year;
    int birth_month;
    int birth_day;
    double height;//--cm
    double weight;//--kg
};//-- do not forget the semi-colon
```

- This code snippet tells the compiler that we are defining a `struct` named `Person`.
- The `Person` `struct` contains 6 data members.
- `Person` is just a definition and no memory is allocated at this time.
- Since C++11, it is possible to give non-static (regular) `struct` members a default value.

Structure | Instances (struct variables)

- To create `struct` variables:

```
Person bjarne;//--1 struct variable
Person guido;//--1 struct variable
```

- You can also declare the instances on the same line.

```
Person bjarne, guido;//--2 struct variables
```

- Memory is allocated only when an instance (a variable) of a structure is created. Typically, the size of a `struct` is the sum of the size of all its members, but not always!
- In our example, the compiler allocates at least 60 bytes for each `struct` variable: 32 bytes for one `std::string`, 12 bytes for three `int`, and 16 bytes for two `double`.
- However, `sizeof{bjarne}` returns 64, due to data structure alignment.

Structure | Member Access

- When we define a variable such as `Employee bjarne`, `bjarne` refers to the entire `struct` (which contains the member variables).
- In order to access the individual members, we use the member selection operator (dot operator).

```
Person bjarne;
bjarne.name = "Bjarne Stroustrup";
bjarne.birth_year = 1950;
bjarne.birth_month = 12;
bjarne.birth_day = 30;
bjarne.height = 165;
bjarne.weight = 65;
```

```
Person guido;
guido.name = "Guido Van Rossum";
guido.birth_year = 1956;
guido.birth_month = 1;
guido.birth_day = 31;
guido.height = 172.72;
guido.weight = 75;
```

- As with normal variables, `struct` member variables are not initialized, and will typically contain garbage data. We must assign a value to each member.

Structure | Member Access

- `struct` variables act like regular variables and you can do any normal operations with them.

```
//--who is younger?
if (bjarne.birth_year > guido.birth_year)
    std::cout << bjarne.name << " is younger than "
    << guido.name << std::endl;
else if (bjarne.birth_year < guido.birth_year)
    std::cout << guido.name << " is younger than "
    << bjarne.name << std::endl;

//--total height of both persons
double total_height = bjarne.height + guido.height;

//--bjarne lost weight
bjarne.weight -= 2;
```

Structure | Uniform Initialization

- As seen in slide 19, assigning values member by member is a little cumbersome.
- C++11 introduced a faster way to initialize `struct`s using a uniform initialization.
- This allows you to initialize some or all the members of a `struct` variable.

```
Person bjarne{"Bjarne Stroustrup", 1950, 12, 30, 165, 65};
Person guido{"Guido van Rossum", 1956, 1, 31, 172.72, 75};
```

- You need to make sure the order of the values corresponds to the order of the members in your `struct`.
- If the initializer list does not contain an initializer for some elements, those elements are initialized to a default value (zero initialization).

```
Person bjarne{"Bjarne Stroustrup", 1950, 12, 30, 165};//--weight=0
```

Structure | Non-static Member Initialization

- Since C++11, it is possible to give non-static (normal) `struct` members a default value.

```
struct Person {
    std::string name{"None"};
    int birth_year{1970};
    int birth_month{1};
    int birth_day{1};
    double height{100};//--cm
    double weight{100};//--kg
};//-- do not forget the semi-colon
```

- You can still use the uniform initialization.

```
Person bjarne{"Bjarne Stroustrup", 1950, 12, 30, 165, 65};
```

- Note: If both initialization types are used, the uniform initialization syntax takes precedence.

Structure | Functions

- A big advantage of using `structs` over individual variables is that we can pass the entire `struct` to a function.

```
void DisplayPerson(Person person){
    std::cout << "Name: " << person.name << std::endl;
    std::cout << "Birth year: " << person.birth_year << std::endl;
    std::cout << "Birth month: " << person.birth_month << std::endl;
    std::cout << "Birth day: " << person.birth_day << std::endl;
    std::cout << "Height: " << person.height << std::endl;
    std::cout << "Weight: " << person.weight << std::endl;
}
int main(){
    Person bjarne{"Bjarne Stroustrup", 1950, 12, 30, 165, 65};
    DisplayPerson(bjarne);
}
```

Structure | Functions

- A function can also return a `struct`, which is one of the few ways to have a function return multiple values at once.

```
Person BuildRandomPerson(){
    return{"Person", 1950, 1, 1, 100, 100};
}
int main(){
    Person person{BuildRandomPerson()};
    DisplayPerson(person);
}
```

Array

- We can use a `struct` to aggregate many different data types into one identifier. This is great for the case where we want to model a single object that has many different properties.
- However, this is not so great for the case where we want to track many related instances of something.
- Consider the case where you want to record the test scores for all students in this class. Without arrays, you would have to allocate 30 almost-identical variables!

```
double testScoreStudent1{};
...
double testScoreStudent18{};
```

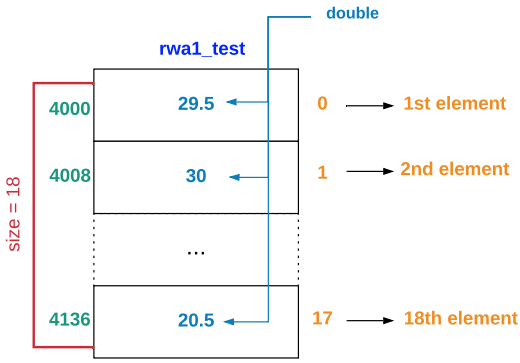
- Arrays give us a much easier way to do this.

```
double testScore[18]{}; //--allocate 18 double in a fixed array
```

Array

- An array is a set of elements of the same data type.
- One can visualize an array as a set of contiguous memory locations. These locations are all of the same size and represent components of the same type.
- We can define arrays of fundamental data types (`int`, `double`, `char`, etc), of STL data types (e.g., `std::string`), or any user-defined data types.
- Definition: An array is a collection of similar type of data in a contiguous memory location under one name.

Array | Characteristics



- Arrays have a fixed size (18).
- Elements of an array are all of the same type (`double`).
- Elements in an array are stored contiguously. The address of the first element (e.g., 4000) is random and the address of the next elements depends on the type of data stored in the array.
- All elements of an array share the same name (rwa1_test).
- Each element of an array is identified by its index. The index is an offset from the beginning of the array.

Array | C and C++ Styles

- Two types of array exist in C++
  - C-style array
    - Feature left over from C (hence the name).
    - Very primitive.
  - C++-style array (since C++11)
    - `std::array` is part of the Standard Library.
    - It comes with many features and benefits.
    - To use C++-style arrays, we need `#include<array>`
    - All methods for C++-style array can be found here.
- **Performance**: You should not notice any difference in runtime performance between C-style and C++-style arrays while you still get to enjoy the extra features of C++-style arrays.

Array | C-style Array | Initialization

- When declaring a fixed array, the length of the array must be known at compile time (compile-time constant).
- Declaration: `type identifier[size]`
- Initialization: `type identifier[size]{list_of_elements}`

```
int array_1[5];//--declaration (contains garbage data)

const int array_length{3};//--a constant
int array_2[array_length]{};//--array with 3 elements

//--With enumerator
enum ArrayElements{
    max_array_length = 4
};
int array_3[max_array_length]{};//--array with 4 elements

#define ARRAY_LENGTH 5//--macro
int array_4[ARRAY_LENGTH]{}; //--syntactically correct but should be avoided
```

Array | C-style Array | Initialization

- There are multiple ways to initialize a C-style array.

```
int array_1[3]{1, 2, 3};//--individual element
int array_2[3]{1, 2};//-- partial initialization

int array_3[]{1, 2, 3};//--size automatically computed

int array_4[3]{};
//--use a range from start up to but not including last
std::fill(array_4, array_3 + 3, 5);//--all elements initialized to 5
std::fill(array_4, array_3 + 2, 5);//--partial initialization
```

- To get the size of a C-style array you can use the function `size()` function from the `iterator` header (`#include <iterator>`)

```
int array_1[3]{1, 2, 3};
std::cout << std::size(array_1) << std::endl;
```

Array | C-style Array | Element Access and Modification

- Subscript notation: `identifier[index]`
- Note: Both C-style and C++-style arrays can use subscript notation [] on indices to access and modify elements.

```
int array_1[]{1, 2, 3};

std::cout << array_1[0] << std::endl;//--1
std::cout << array_1[1] << std::endl;//--2
std::cout << array_1[2] << std::endl;//--3

array_1[2] = 4;
std::cout << array_1[2] << std::endl;//--4
```

- No bounds checking at compile-time. What will happen in the following case at run-time?

```
std::cout << array_1[3] << std::endl;//--error or no error?
```

Array | C-style Array | Element Access and Modification

- The array index out of bounds error is a special case of the buffer overflow error. It occurs when the index used to address array items exceeds the allowed value. It is the area outside the array bounds which is being addressed, and this is why this situation is considered a case of undefined behavior.
- C++ says what should happen if you access the elements within the bounds of an array.
- It is left undefined what happens if you go out of bounds.
- There is a lot that is not specified by the language standard (for a variety of reasons) and this is one of them.
- In general, whenever you encounter undefined behavior, anything might happen.
- There are many cases of undefined behavior in C++

Array | Array Name

- Note: The name of a C-style array represents the address of the first element in that array.
- We will see this in more details in the future.

```
int array_1[]{1, 2, 3};
std::cout << array_1 << std::endl;    //-- 0x00007ffcdb6e8acc
std::cout << &array_1[0] << std::endl;//-- 0x00007ffcdb6e8acc
std::cout << &array_1[1] << std::endl;//-- 0x00007ffcdb6e8ad0
std::cout << &array_1[2] << std::endl;//-- 0x00007ffcdb6e8ad4
```

Array | C++-style Array | Initialization

- C++-style array syntax:

```
std::array<type,size> identifier={list of elements}
```

or

```
std::array<type,size> identifier{list of elements}
```

```
std::array<int, 3> array_1 = {1, 2, 3};//--each element
std::array<int, 3> array_2 = {1};//--partial initialization
```

or

```
std::array<int, 3> array_1 {1, 2, 3};//--each element
std::array<int, 3> array_2 {1};//--partial initialization
```

- Initialize all elements of the array at once.

```
std::array<int, 3> array_1;
array_1.fill(6);//--all elements set to 6
```

Array | C++-style Array | Initialization

- With C-style arrays, the size of an array can be deduced during initialization.

```
int arr_1[]{1,4,5};//--size=3
```

- The size of a C++-style arrays MUST be provided during initialization.

```
//--error: wrong number of template arguments
std::array<int> array_1 {1, 2, 3};
```

- To retrieve the size, you can use the `size()` method.

```
std::array<int,3> array_1 {1, 2, 3};
std::cout << array_1.size() << std::endl;
```

Array | C++-style Array | Element Access and Modification

- To access elements:

  Subscript notation: `identifier[index]`

  at() method: `identifier.at(index)`

```
std::array<int,3> array_1 {1, 2, 3};

std::cout << array_1[0] << std::endl;//--1
std::cout << array_1[1] << std::endl;//--2
std::cout << array_1[2] << std::endl;//--3
std::cout << array_1[3] << std::endl;//--undefined behavior

std::cout << array_1.at(0) << std::endl;//--1
std::cout << array_1.at(1) << std::endl;//--2
std::cout << array_1.at(2) << std::endl;//--3
std::cout << array_1.at(3) << std::endl;//--error
```

Array | C++-style Array | Element Access and Modification

- To modify elements:

  Subscript notation: `identifier[index] = newvalue`

  at() method: `identifier.at(index) = newvalue`

```
std::array<int,3> array_1 {1, 2, 3};

array_1[0] = -1;
array_1[1] = -2;
array_1[2] = -3;
array_1[3] = -4;//--undefined behavior

array_1.at(0) = -5;
array_1.at(1) = -6;
array_1.at(2) = -7;
array_1.at(3) = -8;//--error
```

Array | Summary

- The name of a C-style array represents the location of the first element in the array (index `0`).
- The index represents the offset from the beginning of the array
  - index `0` has an offset of `0`.
  - index `1` has an offset of `1`.
  - ...
- When initializing a C++-style array you cannot omit the number of elements.
- There is no bounds checking on arrays when the subscript notation is used.

Array | Quiz

- What happens when I run the following program?

```
std::array<int> array_1{"alpha","bravo"};
//-----------------------------------------
std::string array_2[4];
std::cout << array_2[3] << std::endl;
//-----------------------------------------
int array_3[4]{1,2,3,4};
std::cout << array_3.size() << endl;
//-----------------------------------------
double array_4[]{2,3.5};
std::cout << array_4[0] << std::endl;
//-----------------------------------------
int array_5[5]{1,3};
std::cout << array_5[2] << std::endl;
//-----------------------------------------
//-- What is the size of the following array?
double array_6[]{1,3,78,4};
```

Array | Exercise #1

- This exercise must be performed twice, using C-style array and C++-style array.
  1. Declare an array of 3 `int` and initialize the array so that all elements are set to 9.
  2. Retrieve the size of the array with the `size()` function and store the result in a variable `size_array`.
  3. Set the value of the first element to 1.
  4. Set the value of the last element to 7.
  5. Display all the array elements in the terminal.

Vector

- Arrays are great, they are efficient and you can access elements very quickly.
- One of the disadvantage of C-style arrays is that you can only access elements using the subscript notation. However, we saw that bounds checking is not performed with this type of notation.
- In some situations you do not know beforehand how many elements you need as this information comes from user inputs:

```
int num{};
std::cout << "Please enter a number: ";
std::cin >> num;
std::array<int, num> array_1{};
```

- Arrays are not suitable in more complex situations: "Use a sensor/camera to keep track of parts ID (`int`) travelling on a conveyor belt".
    - We have no way of knowing how many parts will travel on the conveyor belt.

Vector

- **Solution#1**: Create an array with a size that you are not likely to exceed. Assume that the maximum number of parts travelling on the conveyor belt will never exceed 2,000.

```
std::array<int, 2000> part_id{};
```

  - What happens in the case the number of parts < 2,000?
  - What happens in the case the number of parts > 2,000?
- **Solution#2**: Use a C++ dynamic storage that can grow as new parts are added.
  - One such data structure to deal with this kind of situation is `std::vector`.

Vector | Definition

- C++ vectors are part of the Standard Library (`std::vector`).
- Vectors are same as dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators.
- Vectors can provide bound checking.
- Many operations can be applied to vectors (e.g., sort, reverse, find, etc).
- To use vectors, we need the header `#include <vector>`
- Methods working with vectors can be found at https://en.cppreference.com.

Vector | Declaration

- Declaration: `std::vector<type> identifier`

```cpp
#include <vector>

int main(){
    //--declaring a vector for int elements
    std::vector<int> vect_1;
    //--declaring a vector for double elements
    std::vector<double> vect_2;
}
```

Vector | Initialization

- There are many different ways to initialize vectors in C++
- We will look at the most common ways:
  1. One by one pushing values.
  2. Initializing like arrays.
  3. Specifying size and initializing all values.

Vector | Initialization

1. One by one pushing values.

```
int main() {
    //--create an empty vector of int
    std::vector<int> vector_1;

    //--vect contains 10,20,30 in this order
    vector_1.push_back(10);
    vector_1.push_back(20);
    vector_1.push_back(30);
}
```

Vector | Initialization

2. Initializing like arrays.

```
int main() {
    std::vector<int> vector_1{10,20,30};
}
```

- We initialize the vector with 3 values. We can add or remove elements to and from the vector if needed.

Vector | Initialization

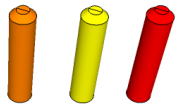
3. Specifying size and initializing all values.

```
int main() {
    int n = 3;

    //--create a vector of size n with all values set to 10
    std::vector<int> vector_1(n, 10);//--parentheses and not curly braces

    //--create a vector of size n with all values set to 0
    std::vector<int> vector_2(n);
}
```

- Can I use `const int n = 3;` instead?
- Should I use `const int n = 3;` instead?

Vector | Element Access

- Subscript notation: `identifier[index]`
- at() method: `identifier.at(index)`



```
std::vector<std::string> peg_colors{"orange","yellow","red"};

std::cout << "1st peg: " << peg_colors[0] << std::endl;
std::cout << "2nd peg: " << peg_colors[1] << std::endl;
std::cout << "3rd peg: " << peg_colors[2] << std::endl;

std::cout << "1st peg: " << peg_colors.at(0) << std::endl;
std::cout << "2nd peg: " << peg_colors.at(1) << std::endl;
std::cout << "3rd peg: " << peg_colors.at(2) << std::endl;
```

Vector | Element Modification

- Subscript notation: `identifier[index] = newvalue`
- at() method: `identifier.at(index) = newvalue`

```
std::vector<std::string> peg_colors{"orange","yellow","red"};

std::cout << "1st peg: " << peg_colors.at(0) << std::endl;
std::cout << "2nd peg: " << peg_colors.at(1) << std::endl;
std::cout << "3rd peg: " << peg_colors.at(2) << std::endl;//--red

peg_colors.at(2)="pink";
std::cout << "3rd peg: " << peg_colors.at(2) << std::endl;//--pink
```

Vector | Adding and Removing Elements

- Vectors size can increase and decrease, unlike arrays.
- To add an element to a vector, the element must be of the appropriate type, e.g., you can only add integer elements to `vector<int>`.
- The vector will automatically allocate the required space.

```
std::vector<std::string> peg_colors{"orange","yellow","red"};

//--orange,yellow,red,pink
peg_colors.push_back("pink");
//--orange,yellow,red,pink,blue
peg_colors.push_back("blue");
```

Vector | Adding and Removing Elements

- One way to delete an element from a vector:
  - Syntax: `vector_name.erase(vector_name.begin()+index)`
  - `begin()` returns an iterator pointing to the first element in the vector (index `0`).

```
std::vector<std::string> alpha_vect{"a","b","c","d","e"};
//--remove the third element
alpha_vect.erase(alpha_vect.begin()+2);
```

Vector | Adding and Removing Elements

- To delete elements within a range:
  - Syntax: `vector_name.erase(first,last)`
  - `first` = `vector_name.begin()+index`
  - `last` = `vector_name.begin()+index`
  - Note: The range includes `first` but not `last` (up to but not including).

```
vector<string> alpha_vect{"a","b","c","d","e"};
//-- Remove the first 3 elements
alpha_vect.erase(alpha_vect.begin(),alpha_vect.begin()+3);
```

```
vector<string> alpha_vect{"a","b","c","d","e"};
//--remove "b", "c", "d"
alpha_vect.erase(alpha_vect.begin()+1,alpha_vect.begin()+4);
```

Vector | Exercise #2

- Initialize a vector of `int` with the numbers: 1000, 100, 10, 1
- Display the size of the vector.
- Display each element of the vector.
- Prompt the user to enter 4 `int` numbers.
- Modify each element of the vector using the values the user entered.
  - First user input is used to modify the first element of the vector.
  - Second user input is used to modify the second element of the vector.
  - ...
- Display each element of the vector using the (at()) method.
- Insert the value 2000 at the end of the vector.
- Delete the 2nd and the 3rd element from the vector.
- Display each element of the vector using `array syntax`.
- Display the size of the vector (number of elements).

Next Class | 09/24

- Reading Material 02 on Flow Control.
- Lecture04: Functions.
- Quiz on Lectures 01, 02, and 03.
- No assignment.
- Stay safe!