The C++

Outline

- Struct, Union and Enum
- Classes and Object
- Constructor and Destructor
- Multifiles
- Preprocessor Directive
- sizeof
- Encapsulation and Setter / Getter
- Pointers

Struct, Union and Enum

1. **Struct (Structure):**

   Groups variables of different data types under one name.

   Used for creating custom data types representing records.

2. **Union:**

   Stores different data types in the same memory location.

   Only one member can be accessed at a time.

   Useful for conserving memory when storing various types of data.

3. **Enum (Enumeration):**

   Assigns names to integral constants.

   Used for better code readability when dealing with related constants.

Struct, Union and Enum

Struct	Union	Enum

// Struct definition
 	
struct Point {
  int x;
  int y;
};
	// Union definition
	
	union Data {
	  int intValue;
	  float floatValue;
	  char stringValue[20];
	};
		// Enum definition
		
		enum DataType {
		  INT_TYPE,
		  FLOAT_TYPE,
		  STRING_TYPE
		};

Classes and Object

1. Classes:

    Blueprint/template for creating objects.

    Define properties and behaviors.

    Encapsulate data and provide methods.

    Can inherit from other classes.

    Can override methods from parent classes to provide specific implementations in subclasses.

2. Objects:

    Instances of classes.

    Have unique identity, attributes, and behaviors.

    Interact with other objects.

Classes, Object, Data and Functions Members

Class	Object

// Declaration
 	
class Car {
Private:
 	
    // Data Members
    std::string brand;
    int year;
 	
Public:
 	
    // Member Functions
    Car(std::string b, int y);
    std::string getBrand();
    int getYear();
};
	// Object
	
	Car myCar("Toyota", 2020);
	
	// Calling Member Function
	
	std::cout << "Class - Car Brand: " <<
	myCar.getBrand() << ", Year: " <<
	myCar.getYear() << std::endl;

Constructor and Destructor

Struct	Class

// Function to initialize a Point object with given coordinates
void initPoint(struct Point* point, int x, int y)
{
point->x = x;
point->y = y;
}
 	
// Function to perform cleanup for a Point object
void cleanupPoint(struct Point* point) {
// Free dynamically allocated memory for the Point object
free(point);
}
	// Constructor
	// Assign Values to Data Memebers
	Car::Car(std::string b, int y) : brand(b), year(y) {}
	
	// Destructor definition
	Car::~Car() {
	// Add any necessary cleanup code here
	}
	// Talk later

Multifiles

Breaks down a large program into smaller, manageable modules.

Each file focuses on specific functionality, making it easier to understand and maintain.

Code written in separate files can be reused in multiple projects or within the same project, reducing redundancy and promoting code reuse.Like **.h** and **.cpp**

Files encapsulate related functionalities, hiding implementation details and exposing only necessary interfaces.

Maintenance tasks such as debugging, testing, and code reviews become more manageable with smaller, modular code units.

.h .cpp files and Preprocessor Directive

car.h	car.cpp

#ifndef CAR_H
#define CAR_H
 	
#include <string>
 	
class Car {
private:
  std::string brand;
  int year;
 	
public:
  Car(std::string b, int y);
  ~Car();
  std::string getBrand();
  int getYear();
};
	#include "car.h"
	
	Car::Car(std::string b, int y) : brand(b), year(y) {}
	
	std::string Car::getBrand() {
	  return brand;
	}
	
	int Car::getYear() {
	  return year;
	}
	
	Car::~Car() {
	  // Add any necessary cleanup code here
	}

sizeof()

size_t **sizeof**(expression);

Size of Struct Point: 8 bytes

Size of Union Data: 20 bytes

Size of Class Car: 48 bytes

Size of Union Data: 20 bytes

Encapsulation and Setter / Getter

Setter	Getter

// Member function to set the brand of the car
void Car::setBrand(std::string b) { brand = b; }
 	
// Member function to set the manufacturing year of the car
void Car::setYear(int y) { year = y;}
	// Member function to get brand
	std::string Car::getBrand() { return brand; }
	
	// Member function to get year
	int Car::getYear() { return year; }

Pointers

Pointers in C++ are variables that store memory addresses, allowing direct access to memory locations and facilitating dynamic memory allocation, efficient array manipulation, and passing parameters by reference.

- The asterisk (*) is used to declare a pointer variable or to dereference a pointer to access the value it points to.
- The ampersand (**&**) is used to get the memory address of a variable (address-of operator).
- (***this**) refers to the current object itself within a member function of a class, allowing access to its members and methods.
- (**this**) is a pointer that points to the current object instance, allowing access to its member variables and methods within member functions.

  In C++, the dot (.) operator is used to access members of an object directly, while the arrow (**->**) operator is used to access members of an object through a pointer to that object.

Pointers and Memory

Allocation	Deallocation

class MyString {
 	
private:
  char* buffer;
 	
public:
 	
  // Constructor
  MyString(const char* str) {
  size_t length = strlen(str);
  buffer = new char[length + 1];
  strcpy_s(buffer, length + 1, str);
}
…
	// Destructor
	MyString::~MyString() {
	  delete[] buffer;
	}

YouTube

https://youtu.be/BL1hZDJvesM