```
//////// Lecture 94: Union Type - I
/*
SLIDES THAT I DONT HAVE ><
This is about Unions in C++ and not unions in C.
- Gives the ability to represent all the memebers in the same memory.
It is just like a structure or a class, but in a structure and class, all members have their own separate memory, but in union, the different members are represented in the same memory.

- this way we can SAVE SPACE!!

- however, unions have several disadvantages:
=> no way to know which type it holds  (which is the active member)
=> if the nested member has a non-default constructor, then that deletes the default constructor of the union. Additionally, the destructor also becomes deleted, so you will have to manually implement these functions in the union. Before C++11, it was illegal for a union to have a member that contains user defined constructors and destructors, but in C++11 it is allowed as long as you provide implementations of the constructors and destructors in the union.
=> Cannot assign objects of user defined types directly to union member. If the union has a user defined type as a member, then you cannot directly initialise it or even assign to it. Instead you have to use a placement new operator.
=> User defined types are NOT DESTROYED IMPLICITLY. So you will have to explicitly call their destructors.
=> cannot have a base class.
=> Cannot inherit from a union.
=> A union cannot contain virtual functions.

These are some of the limitations of unions and C++.
*/

union Test{
```

```cpp
    int x;
    char ch;
    Test(): x{3} , ch{'a'}{
        std::cout << "Constructor invoked!" << std::endl;
    }// error ! Only can initialise 1 variable in the union.
    // Doing this is a compile time error.
};

union Test{
    int x;
    char ch;
    Test(): x{3}{ // constructor
        std::cout << "Constructor invoked!" << std::endl;
    }//this is correct.
    // Destructor
    ~Test(){
        std::cout << "Destructor invoked." << std::endl;
    }
};
// By default, the members of a union are public just like a structure.
int main(){
    Test t;
    std::cout << t.x << std::endl;
    // Let's assign a new value to the other member.
    t.ch = "a";
    // Now the ACTIVE MEMBER in this union is this character.
    // However, there is no way of knowing which is the active member.
    // so sometimes a programmer may read from a non-active member.
    std::cout << t.x << std::endl; // will cause undefined behaviour.
}
// This is one disadvantage of using a union.
// but one advantage is that both these values are being stored in the same memory.
// The size of the union will be equal to the size of its largest member, and that could be used for storage of other members, if you check the size of the union, then the size will be the size of an integer.
int main(){
    Test t;
    std::cout << sizeof(t) << std::endl; // 4
}
// If you store a character value in this union, it will be stored in the same memory  -all the members in this union will have the same storage.
// The size of storage is determined by the largest member in the union.

//////// Lecture 95: Union Type - II
// let's see what happens when we store user defined objects in a union
// consider these structs:
// these structs are just 2 structs that have everything from the rule of 5 implemented
// and we have logged every single call to the console.
// so this way we will know which methods are called automatically.
struct A {
    A() {
        std::cout << __FUNCSIG__ << std::endl;
    }
    ~A() {
        std::cout << __FUNCSIG__ << std::endl;
    }

    A(const A& other) {
        std::cout << __FUNCSIG__ << std::endl;
    }

    A(A&& other) noexcept{
        std::cout << __FUNCSIG__ << std::endl;
    }

    A& operator=(const A& other) {
        std::cout << __FUNCSIG__ << std::endl;
        if (this == &other)
            return *this;
        return *this;
    }

    A& operator=(A&& other) noexcept {
        std::cout << __FUNCSIG__ << std::endl;
        if (this == &other)
            return *this;
        return *this;
    }
};
struct B {

    B() {
        std::cout << __FUNCSIG__ << std::endl;

    }
    ~B() {
        std::cout << __FUNCSIG__ << std::endl;

    }

    B(const B& other) {
        std::cout << __FUNCSIG__ << std::endl ;
    }

    B(B&& other) noexcept {
        std::cout << __FUNCSIG__ << std::endl ;
    }

    B& operator=(const B& other) {
        std::cout << __FUNCSIG__ << std::endl ;
        if (this == &other)
            return *this;
        return *this;
    }

    B& operator=(B&& other) noexcept {
        std::cout << __FUNCSIG__ << std::endl ;
        if (this == &other)
        if (this == &other)
            return *this;
        return *this;
    }
    virtual void Foo(){}

};

union UDT {
    A a ;
    B b ;
};

int main(){
    UDT udt; // THIS gives you an error.
}
// The problem here is because a and b have user defined default constructors the default constructor of the union is DELETED. And we need to call user defined destructor - because a and b have user defined destructors, the default destructor of the union becomes deleted. So we have to implement this manually.
union UDT {
    A a ;
    B b ;
    UDT() {
        // No need to write anything here!
    }
    ~UDT() {

    }
};
int main()
{
    UDT udt; // works now.
    // No call to constructors a and b, so a or b are not implicitly created.
    // If we want to initialise these, then you have to do that manually.
    udt.a = A(); // I am assigning a (union member) to the default instance, but this is NOT RIGHT! Because you are not allowed to assign a member
```

```
          Because you are not allowed to assign a member to an object that has not been created yet.
160         // note that the instance A() have not been created yet.
161         // you can use the assignment operator only if the instances have been created.
162     }
163 
164     // consider a string object within the union
165     union UDT {
166         A a ;
167         B b ;
168         std::string s;
169         UDT() {
170         }
171         ~UDT() {
172         }
173     }
174     int main(){
175         UDT udt;
176         using namespace std::string_literals;
177         // Assign a strigng here:
178         udt.s = "Hello World"s; // use the string literal so there is no need to perform any conversions.
179     }
180     // When you run this, the result will be undefined, because there is no object to which you can assign the string.
181     // see the exit code - the program crashes at line 178.
182 
183 
184     //// PLACEMENT NEW OPERATOR
185     // Therefore, the only way to initialise user defined types inside a union is through the PLACEMENT NEW OPERATOR. Placement new is another form of new. It only initialises the memory but does not allocate. in our case, inside the union, the memory is already allocated and that memory is the memory that is required by the largest member inside the union. And in our case, it is the std::string.
186     // So the memory is already there, it's just that we have to initialise it
```

```
have to initialise it.

int main(){
    UDT udt;
    new (&udt.s) = std::string("Hello World"); // this is the CORRECT APPROACH.
    new (&udt.a) A{};
    // When you are done, you have to manually invoke the destructor calls.
    udt.a.~A();
}
// Unions are useful for certain kinds of applications.
// C++17 introduced a library type called VARIANT - variant can be used as a type safe union, but we will discuss VARIANT in a separate lecture.

```