

# Union And Structs

Module Code: ELEE1147

Module Name: Programming for Engineers

Credits: 15

Module Leader: Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA

# What are Structs and Unions?

- **Structs:** Composite data types that group variables under a single name.
- **Unions:** Similar to structs but share the same memory space for all members.

# Structs

A struct is a user-defined data type in which different data types can be grouped together under a single name.

It is declared using the `struct` keyword, followed by the structure's name and a block of members enclosed in curly braces `{...}`.

```
struct [NameOfStruct] {  
    char var1;  
    int var2;  
    float var3;  
};
```

⇒

```
struct Person {  
    char name[50];  
    int age;  
    float height;  
};
```

# Accessing Members

- **Accessing Members with Pointers**

- For structs and unions, the arrow operator ( `->` ) is used when accessing members through pointers.

```
struct Person {  
    char name[50];  
    int age;  
    float height;  
};  
  
struct Person person1;  
struct Person *personPointer = &person1;  
  
// Using the arrow operator  
personPointer->age = 25;  
  
// Equivalent longhand notation  
(*personPointer).age = 25;
```

# Arrays of Structs and Struct Members

- Array of Structs

- You can create an array of structs to manage multiple records efficiently.

```
struct Student {  
    char name[50];  
    int age;  
    float grades[5]; // Member is an array  
};  
  
int n = 20; // Number of students  
struct Student *class = (struct Student *)malloc(n * sizeof(struct Student));  
  
---  
  
class[0].age = 20;  
strcpy(class[0].name, "John Doe");  
class[0].grades[0] = 90.5;
```

# Unions

Definition:

```
union Data {  
    int intValue;  
    float floatValue;  
    char stringValue[20];  
};
```

- Members share the same memory location.
- Size of union is the size of the largest member.

# Properties of Unions

- **Memory Sharing**

- All members of a union share the same memory space.
- Size of the union is determined by the largest member.

- **Single-Access at a Time**

- Only one member of the union can be accessed at any given time.
- Modifying one member affects the value of the others.

# Type Conversion with Unions

- **Memory Overlay:**
  - Unions store all their members at the same memory location.
  - This means that if you write a value to one member, you are essentially modifying the same memory that is used by other members.
- **Type Interpretation:**
  - When you access a member of the union, you interpret the stored bits according to the type of that member.
  - For example, if you write an integer to a union member and then access another member of type float, you interpret the same bits as a floating-point number.



# Example of Type Conversion

- In this example, a float value (3.14) is stored in converter.floatValue.
- Then, by accessing converter.intValue, you interpret the same memory as an integer.
- This essentially converts the floating-point value to an integer.

```
union TypeConverter {  
    int intValue;  
    float floatValue;  
};  
  
int main() {  
    union TypeConverter converter;  
    converter.floatValue = 3.14;  
    int convertedInt = converter.intValue;  
}
```

## Caution:

- It's crucial to be cautious when using unions for type conversion to avoid undefined behavior or unexpected results.