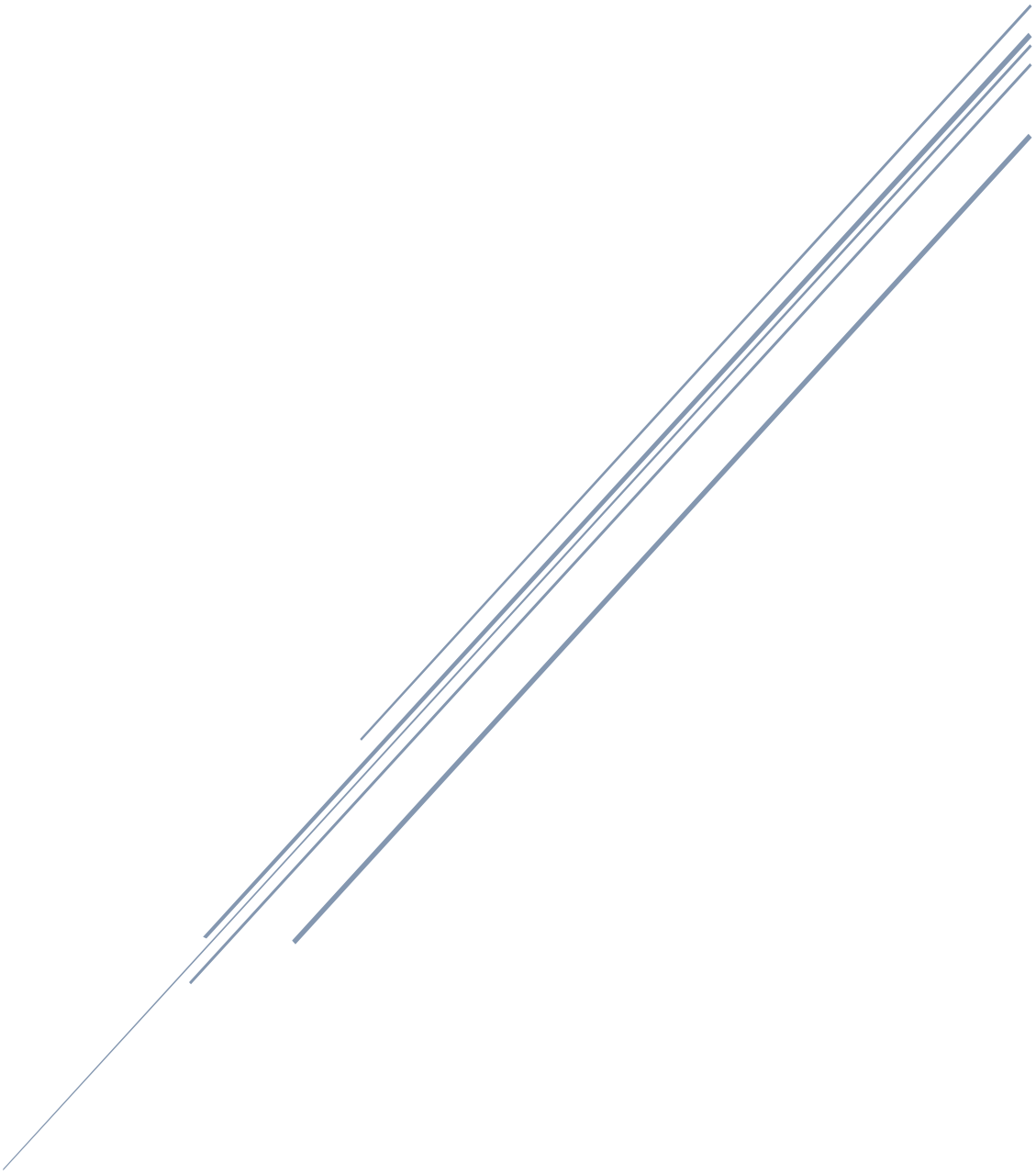
# STRUCTURE AND UNION



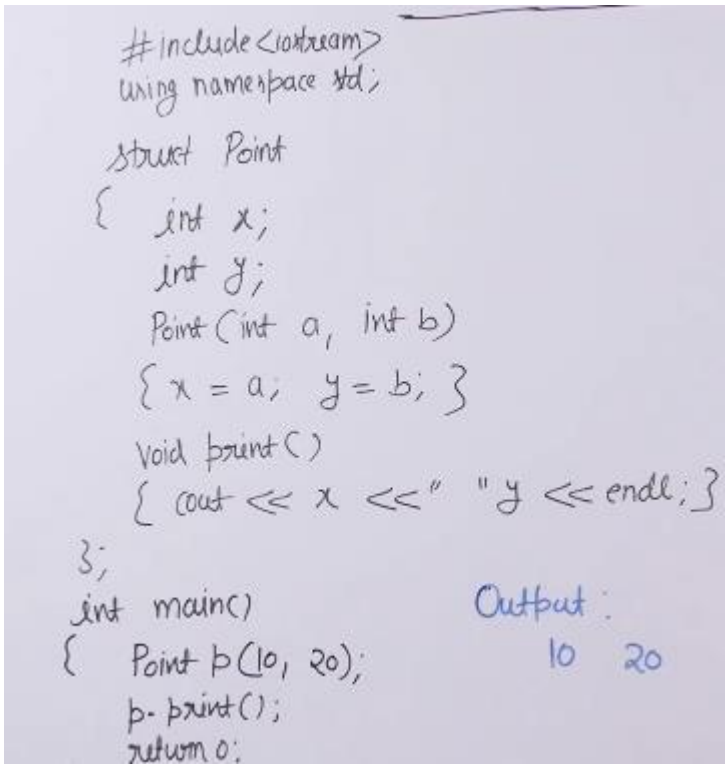
# INDEX

## Table of Contents

# Class vs Struct vs Union in C++

| Class | Struct | Union |
|---|---|---|
| Members of a class are private by default. | Members of a structure are public by default. | Members of a structure are public by default. |
| It is normally used for data abstraction and further inheritance. | It is normally used for the grouping of data | It can be used where the amount of memory used is a key factor. |
| The size of a class is determined by the sizes of its **member variables**, any additional memory required for alignment and **padding**, and the compiler's **memory layout** considerations. | The size is the sum of the sizes of its **individual members**, possibly **padded** for alignment considerations. | The size is determined by the **largest data type** among its members, as all members share the same memory space |

Structure is just like class; one difference is default access specifier. In the below example that struct have same features as class like constructor, functions.



## Class

### Size of Class

```cpp
#include <iostream>
using namespace std;

class DataClass {
public:
    int intValue;
    double doubleValue;
    char charValue;
};

int main() {
    cout << "Size of DataClass: " << sizeof(DataClass) << " bytes" << endl;

    return 0;
}
```

OP

```
Size of DataClass: 16 bytes
```

## Struct

**Remember**

- When we declare a structure or union, we declare a new data type suitable for our purpose. So, **we cannot initialize values as it is not a variable declaration but a data type declaration**.

### Use case

- Store a point in a 2D coordinate system which has two members x and y, which represent coordinate.
- Store student information at one place.
- Store information of webpage, URL, no of views on the page or content of page in the form of string.

```
#include<iostream>
using namespace std;

struct Student
{
    int rollNo;
    string name;
    string address;
};

int main()
{
    // Creating an instance of the Student struct and initializing its members
    Student s = {101, "ABC", "xyz"};

    // Printing the values of the struct members
    cout << s.rollNo << " "
         << s.name   << " "
         << s.address<< "\n";

    return 0;
}
```

OP

```
101 ABC xyz
```

## Designated Initialization

```c
#include<stdio.h>

struct Point
{
    int x;
    int y;
};

int main()
{
    struct Point p = {.y = 10, .x = 20};
    printf("%d %d", p.x, p.y);
    return 0;
}
```

OP

```
20 10
```

**Remember**

If we don't initialize member and accessing members, then we can get some random value. But if we do designated Initialization for some value then other uninitialized value get default value 0.

```c
#include<stdio.h>

struct Point
{
  int x;
  int y;
};

int main()
{
    struct Point p = {.y = 10}; // Initialize y, but x is uninitialized
    printf("%d %d", p.x, p.y);
    return 0;
}
```

OP

```
0 10
```

## Size of Structure

```cpp
#include <iostream>
using namespace std;

struct DataStruct {
    int intValue;
    double doubleValue;
    char charValue;
};

int main() {
    cout << "Size of DataStruct: " << sizeof(DataStruct) << " bytes" << endl;

    return 0;
}
```

OP

Expected output (for a typical 32-bit architecture):

```
Size of DataStruct: 16 bytes
```

Expected output (for a typical 64-bit architecture):

```
Size of DataStruct: 24 bytes
```

**Remember**

- A structure cannot contain a member of its own type because if this is allowed then it becomes impossible for compiler to know size of such struct. Although a pointer of same type can be a member because pointers of all types are of same size and compiler can calculate size of struct

```cpp
struct st {
    int x;
    struct st next;  // This is not possible in C++
};
```

## Structure pointer

**Remember**

The arrow operator -> is used to dereference the pointer and access its members.

```cpp
#include <iostream>

struct Point {
    int x;
    int y;
};

int main() {
    Point myPoint = {3, 7}; // Creating a structure

    // Creating a structure pointer and pointing it to the structure
    Point *ptr = &myPoint;

    // Accessing structure members through the pointer
    std::cout << "x: " << ptr->x << std::endl; // Equivalent to (*ptr).x
    std::cout << "y: " << ptr->y << std::endl; // Equivalent to (*ptr).y

    return 0;
}
```

OP

```
x: 3
y: 7
```

## Union

### Use case

- Type punning, it is a common term used for circumventing the formal type of system in programming languages. We basically convert one type into the other type without any explicit typecasting  and we get the internal representation of that type.

**Remember**

- It allocates memory equal to the largest data type inside it.

Example to show union use same memory location and overwrite for all its members.

```cpp
#include <iostream>
using namespace std;

union UnionExample {
    int intValue;
    int anotherIntValue;
    int yetAnotherIntValue;
};

int main() {
    UnionExample example;

    example.intValue = 42;
    cout << "Int Value: " << example.intValue << endl;

    example.anotherIntValue = 99;
    cout << "Another Int Value: " << example.anotherIntValue << endl;

    cout << "Int Value after Another Int Value assignment: " << example.intValue << endl;
    cout << "Int Value after Another Int Value assignment: " << example.yetAnotherIntValue << endl;

    return 0;
}
```

Total size: 4 bytes

OP

```
Int Value: 42
Another Int Value: 99
Int Value after Another Int Value assignment: 99
Int Value after Another Int Value assignment: 99
```

## Size of Union

```cpp
#include <iostream>
using namespace std;

union DataUnion {
    int intValue;
    double doubleValue;
    char charValue;
};

int main() {
    cout << "Size of DataUnion: " << sizeof(DataUnion) << " bytes" << endl;

    return 0;
}
```

OP

```
Size of DataUnion: 8 bytes
```

## Type punning

The value obtained when performing type punning using unions to interpret a floating-point value as an int can vary across different platforms, compilers, and optimization settings. (Int value can differ)

```cpp
#include <iostream>

int main() {
    union Punning {
        float floatValue;
        int intValue;
    } punning;

    punning.floatValue = 1.1;

    std::cout << "Float value: " << punning.floatValue << std::endl;
    std::cout << "Int value (decimal): " << punning.intValue << std::endl;

    return 0;
}
```

OP

```
Float value: 1.100000
Int value (decimal): 1069547520
```

## Anonymous Union

Using anonymous union to use one type for multiple purposes.

- A Node structure that can be used for both Doubly Linked List and Binary Tree.

```
struct Node {
    int data;
    union {
        struct {
            Node *left, *right;
        };
        struct {
            Node *prev, *next;
        };
    };

    Node(int x) : data(x) {}
};
```

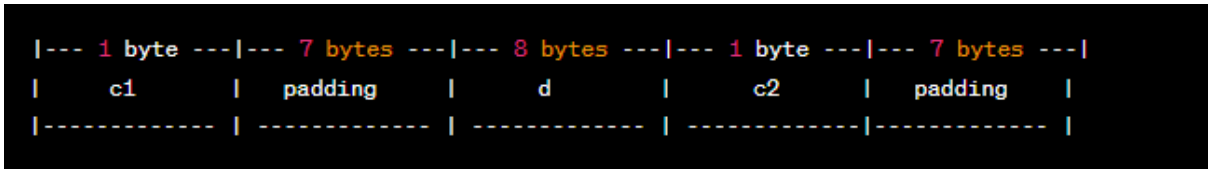
# Structure Alignment and Padding in C++

Structure alignment and padding are closely related concepts. Alignment ensures that **data is accessed efficiently**, and padding is the **extra space inserted between structure members** to achieve that alignment.

**Remember**

- A structure has alignment requirements same as its largest member's requirement.
- Alignment requirements are machine or compiler specific, so we can't read the size required by a structure on a random machine until unless we know individual data type alignment requirement.
- Largest member is **double** in this example. Double require 8-byte alignment requirement. So, 7-byte padded after c1 and c2.

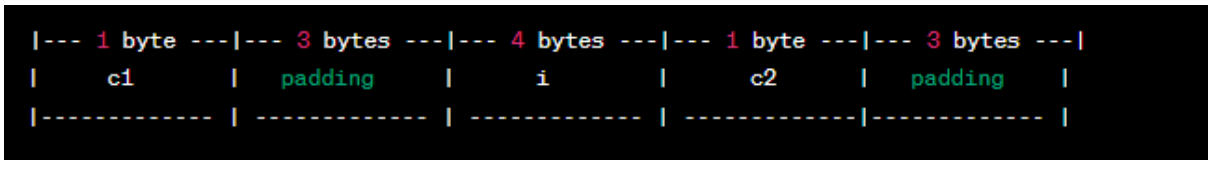
```
struct Example {
    char c1;
    double d;
    char c2;
};
```




Total size: 24 bytes

Largest member is **int** in this example. Int require 4-byte alignment requirement. So, 3-byte padded after c1 and c2.

```
struct Example {
    char c1;
    int i;
    char c2;
};
```




Total size: 12 bytes

**Remember**

By now, it may be clear that padding is unavoidable. There is a way to minimize padding. The programmer should declare the structure members in their increasing/decreasing order of size.

```
struct Example {
    char c1;
    char c2;
    int i;
};
```

```
|--- 1 byte ---|--- 1 byte ---|--- 2 bytes ---|--- 4 bytes ---|
|     c1       |      c2      |    padding    |      i        |
|------------- | ------------ | ------------- | ------------- |
```

Total size: 8 bytes

## Reason for Alignment (Why wasting memory and doing alignment)

- Memory is byte addressable, but CPU read memory in the form of words, In modern computing architectures:
    - 32-bit machines use 4 bytes as their word size.
    - 64-bit machines use 8 bytes as their word size.

  CPU read 4 or 8 bytes at a time to save the CPU cycle to read memory.

- If data is not aligned, then it might across multiple words. In the given example we can see that firstly CPU read 4 byte which contains c1, c2 and 2 bytes of i and in another cycle, CPU read remaining 2 byte. It means CPU need two CPU cycle to read I variable. So that's the reason have alignment so that we can read these things in one word.

```
struct Example {
    char c1;
    char c2;
    int i;
};
```

```
|--- 1 byte ---|--- 1 byte ---|--- 4 bytes ---|
|     c1       |      c2      |      i        |
|------------- | ------------ ' ------------- |
|------------   4 bytes -------------||------------   4 bytes -------------|
```

- We can use for compact representation of structure.
  **__attribute__((packed))** prevent the compiler from adding padding between members.

```
struct Example {
    char c1;
    char c2;
    int i;
} __attribute__((packed));
```

```
|--- 1 byte ---|--- 1 byte ---|--- 4 bytes ---|
|     c1       |      c2      |      i        |
|------------- | ------------ | ------------- |
```

Total size: 6 bytes