C Programming-4

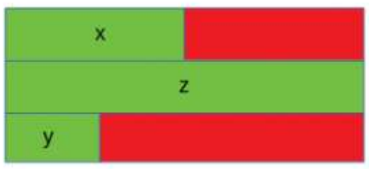
Structures

Structures (also called structs) are a way to group several related variables into one place.
Each variable in the structure is known as a **member** of the structure.
Unlike an array, a structure can contain many different data types (`int`, `float`, `char`, etc.).

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

```
// C program to illustrate
// size of struct
#include <stdio.h>

int main()
{

   struct A {

      // sizeof(int) = 4
      int x;
      // Padding of 4 bytes

      // sizeof(double) = 8
      double z;

      // sizeof(short int) = 2
      short int y;
      // Padding of 6 bytes
   };

   printf("Size of struct: %ld", sizeof(struct A));

   return 0;
}
```

Size: 24



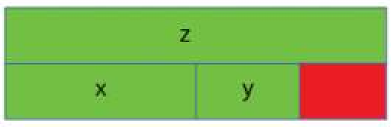
The red portion represents the padding added for data alignment and the green portion represents the struct members. In this case, **x** (int) is followed by **z** (double), which is larger in size as compared to **x**. Hence padding is added after **x**. Also, padding is needed at the end for data alignment.

Case 2:

```
#include <stdio.h>

int main()
{

   struct B {
      // sizeof(double) = 8
      double z;

      // sizeof(int) = 4
      int x;

      // sizeof(short int) = 2
      short int y;
      // Padding of 2 bytes
   };
```

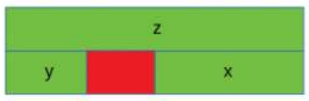
```
    printf("Size of struct: %ld", sizeof(struct B));
    return 0;
}
```

```
Size of struct: 16
```



```
int main()
{

    struct C {
        // sizeof(double) = 8
        double z;

        // sizeof(short int) = 2
        short int y;
        // Padding of 2 bytes

        // sizeof(int) = 4
        int x;
    };

    printf("Size of struct: %ld", sizeof(struct C));

    return 0;
}
```

```
Size of struct: 16
```



Union:

In C, **union** is a user-defined data type that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. Due to this, only one member can store data at the given point in time.

```
union abc{
    int a;
    char b;

};

int main() {
 union abc var;
 var.a=10;
 printf("%d",var.a);
 printf("%c",var.b);

    return 0;
}
```

Output: 10

Both int a and char c share same memory location. Then var.a =10 so a and b occupy 10 .. that is print only 10 but var.b not print anything.

```
#include <stdio.h>
union unionJob
{
   //defining a union
   char name[32];
   float salary;
   int workerNo;
} uJob;

struct structJob
{
   char name[32];
   float salary;
   int workerNo;
} sJob;

int main()
{
   printf("size of union = %d bytes", sizeof(uJob));
   printf("\nsize of structure = %d bytes", sizeof(sJob));
   return 0;
}
size of union = 32
size of structure = 40
```

Why this difference in the size of union and structure variables?

Here, the size of sJob is 40 bytes because

- the size of name[32] is 32 bytes
- the size of salary is 4 bytes
- the size of workerNo is 4 bytes

However, the size of uJob is 32 bytes. It's because the size of a union variable will always be the size of its largest element. In the above example, the size of its largest element, (name[32]), is 32 bytes.

With a union, all members share **the same memory**.

```
#include <stdio.h>
union Job {
   float salary;
   int workerNo;
} j;

int main() {
   j.salary = 12.3;

   // when j.workerNo is assigned a value,
   // j.salary will no longer hold 12.3
   j.workerNo = 100;

   printf("Salary = %.1f\n", j.salary);
   printf("Number of workers = %d", j.workerNo);
   return 0;
}
```

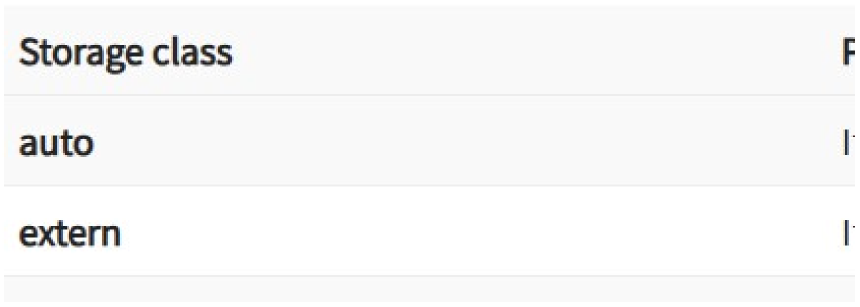
Storage Class:

In C, storage classes define the lifetime, scope, and visibility of variables. They specify where a variable is stored, how long its value is retained, and how it can be accessed which help us to trace the existence of a particular variable during the runtime of a program.

In C, there are **four primary storage classes:**

Table of Content

- auto
- register
- static
- extern



Storage class
auto
extern

auto

This is the default storage class for all the variables declared inside a function or a block. Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope).

Properties of auto Variables

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** RAM
- **Lifetime:** Till the end of its scope

```
int main() {

   // auto is optional here, as it's the default storage class
   auto int x = 10;
   printf("%d", x);
   return 0;
}
Output: 10
```

```
#include <stdio.h>
int main( )
{
  auto int j = 1;
  {
    auto int j= 2;
    {
      auto int j = 3;
      printf ( " %d ", j);
    }
    printf ( "\t %d ",j);
  }
  printf( "%d\n", j);}
```

Output: 3 2 1

Explanation: The **auto** keyword is used to declare a local variable with automatic storage. However, in C, local variables are automatically auto by default, so specifying auto is optional.
auto keyword is not used in front of functions as functions are not limited to block scope.

static

This storage class is used to declare static variables that have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

Properties of static Storage Class

- **Scope:** Local
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program

```
void counter() {

   // Static variable retains value between calls
   static int count = 0;
   count++;
   printf("Count = %d\n", count);
}
int main() {

   // Prints: Count = 1
   counter();

   // Prints: Count = 2
   counter();
   return 0;
}
```

Output:

```
Count = 1

Count = 2
```

Explanation: The static variable count retains its value between function calls. Unlike local variables, which are reinitialized each time the function is called, static variables remember their previous value.

register

This storage class declares register variables that have the same functionality as that of the auto variables. The only difference is that the compiler tries to store these variables in the register of the microprocessor if a free register is available making it much faster than any of the other variables.

Properties of register Storage Class Objects

- **Scope:** Local
- **Default Value:** Garbage Value
- **Memory Location:** Register in CPU or RAM
- **Lifetime:** Till the end of its scope

```
int main() {

   // Suggest to store in a register
   register int i;
   for (i = 0; i < 5; i++) {
      printf("%d ", i);
   }
   return 0;
}
```

```
Output: 0 1 2 3 4
```

Explanation: The register keyword suggests storing the variable in a register for faster access, but the compiler may not always honor this request based on available CPU registers.

extern

Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used. Basically, the value is assigned to it in a different block and this can be overwritten/changed in a different block as well.
Also, a normal global variable can be made extern as well by placing the ‘extern’ keyword before its declaration/definition in any function/block.

Properties of extern Storage Class Objects

- **Scope:** Global
- **Default Value:** Zero
- **Memory Location:** RAM
- **Lifetime:** Till the end of the program.

For example, the below file contains a global variable.