Type: Set of values and operations.

```
Struct {        struct {        tA  O1
  int a;          int a           tB  O2
  int b;          int b
} tA            } tB            O1 = O2  ← Can be assigned if they are same fields.
```

```
typedef int feet
typedef int meter
    feet O1, meter O2
    O1 = O2  ← is valid
```

```
int a[3];
int b[3];
```

a = b ← not valid

'a' returns a Const. pointer which can't be changed.

```
enum Color {RED, GREEN, BLUE};
```

↓ same as

```
#define RED 0
#define GREEN 1
```

float has less number of values than int

Floating point Arithmetic is not Commutative and associated.

↓

Due to format in which the floating point numbers are stored and represented, it rounds off of the numbers during Caluclations.

c = a

c = c + 0.5

d = b

d = d + 0.3

c = c - 0.5

d = d - 0.3

if(a+b == c+d) {S1}

else {S2}

⇓

Here S_2 will get executed due to floating point exception.

Valid check: if($|(a+b) - (c+d)| \leq \epsilon$) {S1}

else {S2}

→ a%2 == 0 is faster in many compilers than checking it with any other value.

```
class animal {
  char* name;
  char getname();
};
```

```
class dog : public animal
{
  enum class breed { ... }
  breed getbreed()
}
```

Subtype

Supertype

animal

↓

dog

```
Class A {
    int a;
}
```

int x * X *

```
Class B : public A {
    int b;
}
```

int x int x *

```
class C : public B {
    int c;
}
```

int x int x int

A is supertype of B and C.

We can talk about supertype and subtype only in case of inheritance.

→ type widening (safe) — x = y — type(x) > type(y).

→ type narrowing — y = x — type(x) > type(y).

(unsafe)

type x is supertype of y.

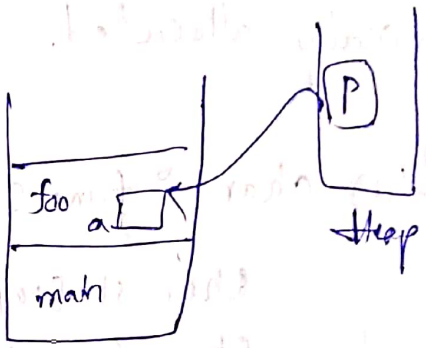
```
foo(const int a)
{
    ---------
}
```

Is Const int is sub/super type of int?

```
int b=5;
foo(b);
```

Garbage Collection

```
int *p;
foo() {
int a = 5;
p = &a;
}
int main() {
int b;
foo();
b = *p;  → dangling pointer.
}  → not a Segmentation fault.
```



After call in main()
foo call stack gets cleared

Memory Leak: occurs due to not freeing the new elements allocated in heap.

Disadvantages:
→ Performance Overhead.

Rust
↓
No Run-time Overhead.
RAII

Shared-ptr.

```
float a = 5.0
switch(a) {
case 5.0:
    S1
case 5.1:
    S1
}
```

Switch
float is only defined for integer data-types.
Jump.
Assembly Creates a Table for switch values

"Constexpr"

int a,b,c;

'IOCCC'

'Underhanded C Code Contest'

'MISRA C' std.

```
Switch(a) {
 Case 1:
     if(a>0){
 Case 2:
       a=3
       break;
     }
}
```

⇒ It's not allowed in 'C'.

doesn't allow

Switch is faster than if...else.

① int a[100];
for (i=0;i<100;i++):

a[i]=a[i-1]+1

for(auto i:a)
S

② int a[100]

for each k in a{
...
}

In ① We can't write Sty like a[i]=a[i-1]+1 as we only have access to elements. We cant skip elements by ② methods.

try, catch and rethrow.

→ Exception handling should always be used to handle the Situation that are beyond user's Control.

```
int x=1;
Switch(x){
 x+=1;
Case 1 : Print(Choice is 1);}
```

⇒ >> choice is 1.
After switch statement, Control transfers to matching case, the statements written before are not executed.

Scanned by CamScanner

→ A dangling pointer is one that has a value (not NULL) which refers to some memory which is not valid for the type of object you expect. For example if you set a pointer to an object then overwrote that memory with Stg. else unrelated or freed the memory if it was dynamically allocated.

Eg: ① char *func() {

```
    char str[10];
    strcpy(str, "Hell");
    return str;
}
//Returned Pointer to str has gone out of scope after call.
void main() {
cout << func();
}
```

↓

Segmentation fault.

② int *c = malloc(sizeof(int));

```
free(c);
*c = 3; //Writing to freed location (error)
```

→ A memory leak is memory which hasn't been freed, there is no way to access it (or free it) now, as there are no ways to get to it anymore.

eg: Void func() {

Char *ch = malloc(10);

}

// ch is not valid outside, no way to access malloc-ed memory.

→ GC works in a way such that the runtime detects unused objects and object graphs in the background. They keep track when objects are no longer referenced in the code.

→ All object invariants are pre condition and post condition to all member methods except for constructors and destructors. For constructor the invariant is just a post condition, for destructor invariant is just a pre condition.

```
int* p;
void
{ int a=5
  p=&a
}

printf(*p)
```

Enum

enum variable

Decl: enum days { sun, Mon, Tue, Wed, Thu, fri, sat };

↓ keyword ↓ state=0 ↓ state=1 ...

Instantiation: enum days day;

operation: day = wed

Printf("%d", day) >> 3

Int i; for (i=sun; i<=sat; i++)

Printf("%d", i) >> 0 1 2 3 4 5 6

-> Default assignment of enum starts from '0'.

-> We can assign values to some name in any order. All unassigned names get value as value of previous name plus one.

-> Value assigned to enum names must be some integral constant.

-> All enum constants must be unique in their scope.

enum state { working, failed };

enum result { failed, passed };

>> Compilation error.

-> Integral expressions used in switch labels must be Const. expressions

int x=2; int arr[] = {1,2,3};

Switch (x) {

Case arr[0]: S1

Case arr[1]: S2

Case arr[2]: S3

}

=> Compilation Error: Case label doesn't reduce to integer constant.

$2^{**}3^{**}4$ is not allowed in fortran. → Precedence and associativity Comply for infix ops

→ Expressions Provide Value to Surrounding text.

→ Imperative languages Compute by means of Side-Effects
- A future Computation is influenced in a way other than returning a Value to Surrounding Context.

→ functional languages have no Side-Effects. The value of an Expression is independent of time it is evaluated.

→ C allows expressions to appear where statements are expected.

→ Some languages require definite assignment, they Complain if they see no initialization at Compile time.

→ Using Commutativity in addn of floating point values Can be dangerous.

→ Short circuit evaluation.

→ Lisp and Ruby provide throw... catch mechanisms

- Sequencing
  - Side Effect may be desired for Subroutines
  - rand() returns a different number each time called.

→ Jump tables make Sense when the range of options are dense since they can be efficiently packed into an array whose index is the Selector's value and whose Contents are the addresses of branch options.

→ Enumeration-controlled loop: Values over a finite set.

→ Logically-controlled loop: based on monitoring boolean condition.

→ Operation/function/expression is said to have a side effect if it modifies some state variable(s) outside its local environment, that is to say has an observable effect besides returning a value to the invoker of it.

→ C's loops are logically controlled, but they can simulate enumeration controlled.

→ Machine and Assembly languages have no types.

→ Structural equivalence specifics depend on language.

→ A Compiler can determine structural equivalence by expanding their definitions recursively until all is left is a string of type constructors, field names and built-in types.

→ Coercion is using some type where something else is expected.
- It is a form of implicit conversion.
- It often requires dynamic run-time checks.