OTHER DATA TYPES

Type Aliases (typedef / using)

In C++, any valid type can be aliased so that it can be referred to with a different identifier.

```
1 typedef char C;
2 typedef unsigned int WORD;
3 typedef char * pChar;
4 typedef char field [50];
```

Once these aliases are defined, they can be used in any declaration just like any other valid type:

```
1 C mychar, anotherchar, *ptc1;
2 WORD myword;
3 pChar ptc2;
4 field name;
```

More recently, a second syntax to define type aliases was introduced in the C++ language:

```
using new_type_name = existing_type ;
```

For example, the same type aliases as above could be defined as:

```
1 using C = char;
2 using WORD = unsigned int;
3 using pChar = char *;
4 using field = char [50];
```

The only difference being that **typedef** has certain limitations in the realm(alan) of templates that **using** has not.

Neither **typed** nor **using** create new distinct data types. They only create synonyms of existing types. That means that the type of **myword** above, declared with type WORD, can as well be considered of type unsigned int; it does not really matter,

since both are actually referring to the same type.

Type aliases can be used **to reduce the length of long or confusing type names**, but they are most useful as tools **to abstract programs from the underlying types they use.** For example, by using an alias of **int** to refer to **a particular kind of parameter** instead of using **int** directly, it allows for the type to **be easily replaced by long** (or some other type) in a later version, without having to change every instance where it is used.

Unions **(**One portion of memory to be accessed as different data types.**)**

```
union type_name {
   member_type1 member_name1;
   member_type2 member_name2;
   member_type3 member_name3;
   .
   .
} object_names;
```

This creates a new union type, identified by **type_name,** in which all its member elements **occupy the same physical space in memory.**
The size of this type is the one of the largest member element.

```
1 union mytypes_t {
2   char c;
3   int i;
4   float f;
5 } mytypes;
```

declare an object (**mytypes**) with three members. Each of these members is of a different data type.

```
mytypes.c
mytypes.i
mytypes.f
```

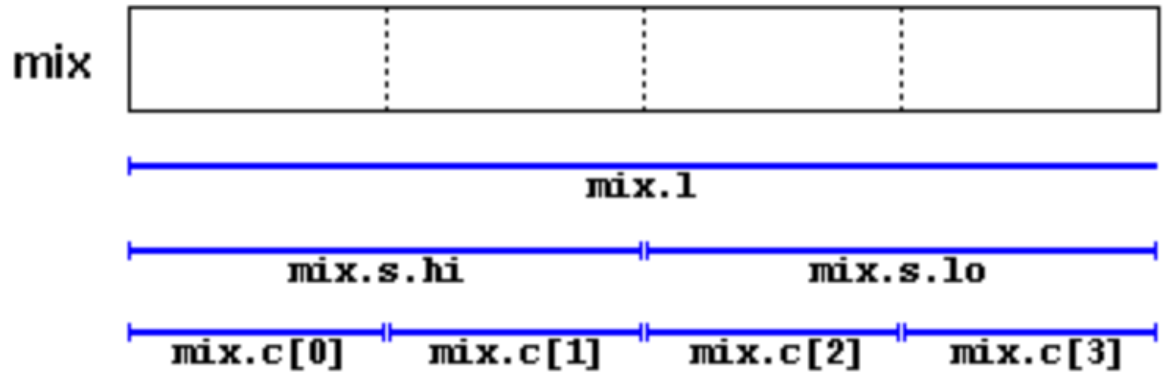
But since **all of them are referring to the same location in memory,** the modification of one of the members will **affect the value of all of them**.
It is not possible to store different values in them in a way that each is independent of the others.

One of the uses of a union is **to be able to access a value either in its entirety(bütün) or as an array or structure of smaller elements**.

```
union mix_t {
  int l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

If we assume that the system where this program runs has an **int type with a size of 4 bytes**, and a **short type of 2 bytes**, the union defined above allows the **access to the same group of 4 bytes**: mix.l, mix.s and mix.c, and which we can use according to how we want to access these bytes:
as if they were a single value of type int, or **as if they were two values of type short**, or **as an array of char elements**, respectively.
The example mixes types, arrays, and structures in the union **to demonstrate different ways to access the data.**

For a **little-endian system**, this union could be represented as:



The exact alignment and order of the members of a union in memory **depends on the system**, with the **possibility of creating portability issues.**

Anonymous Unions

When **unions** are members of a class (or structure), they can be declared with **no name.** In this case, they become **anonymous unions,** and its members are **directly accessible from objects by their member names.**

structure with regular union	structure with anonymous union

`struct book1_t {`
`  char title[50];`
`  char author[50];`
`  union {`
`  float dollars;`
`  int yen;`
`  } price;`
`} book1;`
	`struct book2_t {`
	`  char title[50];`
	`  char author[50];`
	`  union {`
	`  float dollars;`
	`  int yen;`
	`  };`
	`} book2;`

In the first one, the member union has a name(**price**), while in the second it has not. This affects the way to access members **dollars** and **yen** of an object of this type. For an object of the **first type** (with a regular union), it would be:

```
1 book1.price.dollars
2 book1.price.yen
```

whereas for an object of the second type (which has an **anonymous union**) it would be:

```
1 book2.dollars
2 book2.yen
```

Remember that because **it is a member union (not a member structure)**, the members **dollars** and **yen** actually **share the same memory location**, so **THEY CANNOT BE USED TO STORE TWO DIFFERENT VALUES SIMULTANEOUSLY.** The price can be set in **dollars or in yen, but not in both simultaneously.**

Enumerated Types (enum) (Defined with a set of custom identifiers, known as ***enumerators***, as possible values.)
Objects of these ***enumerated types*** can take any of these enumerators as value.
instantiate = savı örnek vererek desteklemek, kanıt sunmak, kanıt göstererek bir iddiada bulunmak

A new type of variable called **colors_t** could be defined **to store colors with the following declaration.**

```
enum colors_t {black, blue, green, cyan, red, purple, yellow, white};
```

This declaration **includes no other type, neither fundamental nor compound,** in its definition.
This **creates a whole new data type** from scratch(boş,gelişigüzel) without basing it on any other existing type.
Once the **colors_t** enumerated type is declared, the following expressions will be valid:

```
colors_t mycolor;

mycolor = blue;
if (mycolor == green) mycolor = red;
```

Values of ***enumerated types*** declared with enum are implicitly(dolaylı olarak) convertible to the integer type **int**, and vice versa.
In fact, the elements of such an enum **are always assigned an integer numerical equivalent internally**, of which **they become an alias**.

If it is not specified otherwise, **the integer value equivalent to the first possible value is 0**, the equivalent to **the second is 1**, to **the third is 2**, and so on...
Therefore, in the data type **colors_t** defined above, black would be equivalent to 0, blue would be equivalent to 1, green to 2, and so on...

```
enum months_t { january=1, february, march, april,
                may, june, july, august,
                september, october, november, december} y2k;
```

In this case, the **variable y2k** of the enumerated type **months_t** can contain any of the 12 possible values that go from **january** to **december** and that are equivalent to the values between **1 and 12 (not between 0 and 11, since january has been made equal to 1)**.

Because enumerated types declared with enum are **implicitly convertible to int**, and **each of the enumerator values is actually of type int**,
THERE IS NO WAY TO DISTINGUISH 1 FROM JANUARY - THEY ARE EXACT SAME VALUE OF THE SAME TYPE.

Enumerated Types With Enum Class
It is possible to create real enum types that are **neither implicitly convertible to**

int and that **neither have enumerator values of type int**, but of the enum type itself, thus **preserving type safety**.
They are declared with **enum class** (or enum struct) instead of just **enum.**

```
enum class Colors {black, blue, green, cyan, red, purple, yellow, white};
```

Each of the enumerator values of an enum class type **needs to be scoped into its type** (this is actually also possible with enum types, but it is only optional).

```
1 Colors mycolor;
2
3 mycolor = Colors::blue;
4 if (mycolor == Colors::green) mycolor = Colors::red;
```

————
Enumerated types declared with enum class also **have more control over their underlying type**; it may be any integral data type, such as char, short or unsigned int, which essentially serves to determine the size of the type.
This is specified by a **colon(:)** and the underlying type following the enumerated type.

```
enum class EyeColor : char {blue, green, brown};
```

Eyecolor is a distinct type with the same size of a char (1 byte).