Compound Data Types:

# 14. Other Data Types

## Defined data types (typedef)

C++ allows the definition of our own types based on other existing data types. We can do this using the keyword `typedef`, whose format is:

```
typedef existing_type new_type_name ;
```

where `existing_type` is a C++ fundamental or compound type and `new_type_name` is the name for the new type we are defining. For example:

```
typedef char C;
typedef unsigned int WORD;
typedef char * pChar;
typedef char field [50];
```

In this case we have defined four data types: `C`, `WORD`, `pChar` and `field` as `char`, `unsigned int`, `char*` and `char[50]` respectively, that we could perfectly use in declarations later as any other valid type:

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
```

`typedef` does not create different types. It only creates synonyms of existing types. That means that the type of `myword` can be considered to be either `WORD` or `unsigned int`, since both are in fact the same type.

`typedef` can be useful to define an alias for a type that is frequently used within a program. It is also useful to define types when it is possible that we will need to change the type in later versions of our program, or if a type you want to use has a name that is too long or confusing.

## Unions

Unions allow one same portion of memory to be accessed as different data types, since all of them are in

fact the same location in memory. Its declaration and use is similar to the one of structures but its functionality is totally different:

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

All the elements of the `union` declaration occupy the same physical space in memory. Its size is the one of the greatest element of the declaration. For example:

```
union mytypes_t {
  char c;
  int i;
  float f;
  } mytypes;
```

defines three elements:

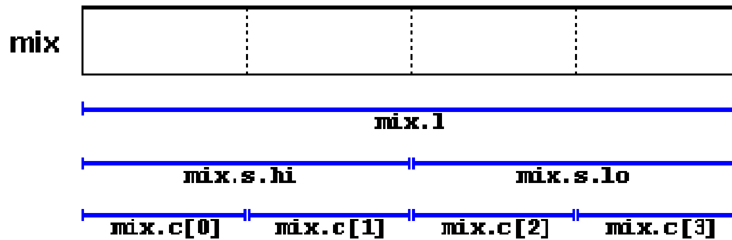
```
mytypes.c
mytypes.i
mytypes.f
```

each one with a different data type. Since all of them are referring to the same location in memory, the modification of one of the elements will affect the value of all of them. We cannot store different values in them independent from each other.

One of the uses a union may have is to unite an elementary type with an array or structures of smaller elements. For example:

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

defines three names that allow to access 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 `long`-type data, as if they were two `short` elements or as an array of `char` elements, respectively. I have mixed types, arrays and structures in the union so that you can see the different ways that we can access the data. For a *little-endian* system (most PC platforms), this union could be represented as:




The exact alignment and order of the members of a union in memory is platform dependant. Therefore be aware of possible portability issues with this type of use.

## Anonymous unions

In C++ we have the option to declare anonymous unions. If we declare a union without any name, the union will be anonymous and we will be able to access its members directly by their member names. For example, look at the difference between these two structure declarations:

<table>
<tr><th>structure with regular union</th><th>structure with anonymous union</th></tr>
<tr><td><pre>struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  } price;
} book;</pre></td><td><pre>struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  };
} book;</pre></td></tr>
</table>

The only difference between the two pieces of code is that in the first one we have given a name to the union (`price`) and in the second one we have not. The difference is seen when we access the members `dollars` and `yens` of an object of this type. For an object of the first type, it would be:

```
book.price.dollars
book.price.yens
```

whereas for an object of the second type, it would be:

```
book.dollars
book.yens
```

Once again I remind you that because it is a union and not a struct, the members `dollars` and `yens` occupy the same physical space in the memory so they cannot be used to store two different values simultaneously. You can set a value for `price` in dollars or in yens, but not in both.

## Enumerations (enum)

Enumerations create new data types to contain something different that is not limited to the values

fundamental data types may take. Its form is the following:

```
enum enumeration_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

For example, we could create a new type of variable called `color` to store colors with the following declaration:

```
enum colors_t {black, blue, green, cyan,

red, purple, yellow, white};
```

Notice that we do not include any fundamental data type in the declaration. To say it somehow, we have created a whole new data type from scratch without basing it on any other existing type. The possible values that variables of this new type `color_t` may take are the new constant values included within braces. For example, once the `colors_t` enumeration is declared the following expressions will be valid:

```
colors_t mycolor;

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

Enumerations are type compatible with numeric variables, so their constants are always assigned an integer numerical value internally. If it is not specified, the integer value equivalent to the first possible value is equivalent to `0` and the following ones follow a +1 progression. Thus, in our data type `colors_t` that we have defined above, `black` would be equivalent to `0`, `blue` would be equivalent to `1`, `green` to `2`, and so on.

We can explicitly specify an integer value for any of the constant values that our enumerated type can take. If the constant value that follows it is not given an integer value, it is automatically assumed the same value as the previous one plus one. For example:

```
enum months_t { january=1, february, march,

    april, may, june, july,

august, september, october,

    november, december} y2k;
```

In this case, variable `y2k` of enumerated type `months_t` can contain any of the 12 possible values that go from

`january` to `december` and that are equivalent to values between `1` and `12` (not between `0` and `11`, since we have made `january` equal to `1`).