**Enumerated types (enum).**

*Enumerated types* are types that are 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.

Their syntax is:

|  |
| --- |
| enum type\_name {  value1,  value2,  value3,  .  .  } object\_names; |

This creates the type *type\_name*, which can take any of *value1*, *value2*, *value3*, ... as value. Objects (variables) of this type can directly be instantiated as *object\_names*.

For example, a new type of variable called *colors\_t* could be defined to store colours with the following declaration:

|  |  |  |
| --- | --- | --- |
|  | enum colors\_t {black, blue, green, cyan, red, purple, yellow, white}; |  |

Notice that this declaration includes no other type, neither fundamental nor compound, in its definition. To say it another way, somehow, this creates 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 enumerators listed within braces. For example, once the *colors\_t* enumerated type is declared, the following expressions will be valid:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 | colors\_t mycolor;    mycolor = blue;  if (mycolor == green) mycolor = red; |  |

Values of *enumerated types* declared with *enum* are implicitly convertible to an integer type. In fact, the elements of such an *enum* are always assigned an integer numerical equivalent internally, to which they can be implicitly converted to. 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...

A specific integer value can be specified for any of the possible values in the enumerated type. And if the constant value that follows it is itself not given its own value, it is automatically assumed to be the same value plus one. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | 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).

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  18  19  20  21 | #include <iostream>  using namespace std;  enum K8  {  k,  h,  x,  K //or K, (with comma)  } object4;  K8 object1 = k;  K8 object2 = h;  K8 object3 = x;  int main(void)  {  object4 = K;  cout << "The numerical value of k: " << k << endl;  cout << "The numerical value of h: " << object2 << endl;  cout << "The numerical value of x: " << x << endl;  cout << "The numerical value of K: " << object4 << endl;  cout << "sizeof(K8): " << sizeof(K8) << endl;  } | The numerical value of k: 0  The numerical value of h: 1  The numerical value of x: 2  The numerical value of K: 3  sizeof(K8): 4 | [Edit & Run](https://www32.cplusplus.com/doc/tutorial/pointers/) |