**Unions.**

Unions allow one portion of memory to be accessed as different data types. Its declaration and use are similar to the one of structures, but its functionality is totally different:

|  |
| --- |
| 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. For example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 | union mytypes\_t {  char c;  int i;  float f;  } mytypes; |  |

declares an object (*mytypes*) with three members:

|  |  |  |
| --- | --- | --- |
| 1 2 3 | mytypes.c  mytypes.i  mytypes.f |  |

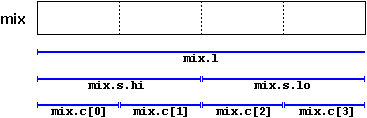
Each of these members is of a different data type. But since all of them are referring to the same location in memory, the modification of one of the members will affect the value of all of them. It is not possible to store different values in them in a way that each is independent of the others.

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

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 | 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17  18  19  20 | #include <iostream>  using namespace std;  union UNION  {  char k;  int h;  double x;  } Union;  void function(void)  {  cout << sizeof(Union.k) << endl;  cout << sizeof(Union.h) << endl;  cout << sizeof(Union.x) << endl;  cout << sizeof(Union) << endl;  cout << sizeof(UNION) << endl;  }  int main(void)  {  function();  } | 1  4  8  8  8  Press any key to continue . . . | [Edit & Run](https://www32.cplusplus.com/doc/tutorial/pointers/) |