Union type

From Wikipedia, the free encyclopedia
(Redirected from Union (computer science))

This article **does not cite any references or sources**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed. *(August 2009)*

In computer science, a **union** is a value that may have any of several representations or formats; or it is a data structure that consists of a variable that may hold such a value. Some programming languages support special data types, called **union types**, to describe such values and variables. In other words, a union type definition will specify which of a number of permitted primitive types may be stored in its instances, e.g., "float or long integer". Contrast with a record (or structure), which could be defined to contain a float *and* an integer; in a union, there is only one value at any given time.

A union can be pictured as a chunk of memory that is used to store variables of different data types. Once a new value is assigned to a field, the existing data is overwritten with the new data. The memory area storing the value has no intrinsic type (other than just bytes or words of memory), but the value can be treated as one of several abstract data types, having the type of the value that was last written to the memory area.

In type theory, a union has a sum type; this corresponds to disjoint union in mathematics.

Depending on the language and type, a union value may be used in some operations, such as assignment and comparison for equality, without knowing its specific type. Other operations may require that knowledge, either by some external information, or by the use of a tagged union.

Untagged unions [edit]

Because of the limitations of their use, untagged unions are generally only provided in untyped languages or in a type-unsafe way (as in C). They have the advantage over simple tagged unions of not requiring space to store a data type tag.

The name "union" stems from the type's formal definition. If a type is considered as the set of all values that that type can take on, a union type is simply the mathematical union of its constituting types, since it can take on any value any of its fields can. Also, because a mathematical union discards duplicates, if more than one field of the union can take on a single common value, it is impossible to tell from the value alone which field was last written.

However, one useful programming function of unions is to map smaller data elements to larger ones for easier manipulation. A data structure consisting, for example, of 4 bytes and a 32-bit integer, can form a union with an unsigned 64-bit integer, and thus be more readily accessed for purposes of comparison etc.

Unions in various programming languages [edit]

C/C++ [edit]

In C and C++, untagged unions are expressed nearly exactly like structures (structs), except that each data member begins at the same location in memory. The data members, as in structures, need not be primitive values, and in fact may be structures or even other unions. However, C++ does not allow for a data member to

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Deutsch
Español
日本語
Polski
Русский
中文
Edit links

be any type that has a full-fledged constructor/destructor and/or copy constructor, or a non-trivial copy assignment operator. For example, it is impossible to have the standard C++ string as a member of a union. (C++11 lifts some of these restrictions.)

Like a structure, all of the members of a union are by default public. The keywords `private`, `public`, and `protected` may be used inside a structure or a union in exactly the same way they are used inside a class for defining private, public, and protected member access.

The primary use of a union is allowing access to a common location by different data types, for example hardware input/output access, perhaps bitfield and word sharing. Unions also provide crude polymorphism. However, there is no checking of types, so it is up to the programmer to be sure that the proper fields are accessed in different contexts. The relevant field of a union variable is typically determined by the state of other variables, possibly in an enclosing struct.

One common C programming idiom uses unions to perform what C++ calls a **reinterpret_cast**, by assigning to one field of a union and reading from another, as is done in code which depends on the raw representation of the values. A practical example is the method of computing square roots using the IEEE representation. This is not, however, a safe use of unions in general.

> Structure and union specifiers have the same form. [. . .] The size of a union is sufficient to contain the largest of its members. The value of at most one of the members can be stored in a union object at any time. A pointer to a union object, suitably converted, points to each of its members (or if a member is a bit-field, then to the unit in which it resides), and vice versa.
>
> —ANSI/ISO 9899:1990 (the ANSI C standard) Section 6.5.2.1

Anonymous union [edit]

Unions can also be anonymous; that is, they do not have a name. Their data members are accessed directly. In addition to this, they have certain other restrictions like:

- They must also be declared as static if declared in file scope. If declared in local scope, they must be static or automatic.
- They can have only public members; private and protected members in anonymous unions generate errors.
- They cannot have function members.

Simply omitting the class-name portion of the syntax does not make a union an anonymous union. For a union to qualify as an anonymous union, the declaration must not declare an object. Example:

```cpp
// anonymous_unions.cpp
#include <iostream>
using namespace std;
int main() {
   union {
      int d;
      char f;
   };

   d = 4;
   cout << d << endl;

   f = 'i';
   cout << f << endl;
}
```

Anonymous unions are supported by C11 and as a non-standard GCC extension.

COBOL [edit]

In COBOL, union data items are defined in two ways. The first uses the `RENAMES` (66 level) keyword, which effectively maps a second alphanumeric data item on top of the same memory location as a preceding data item. In the example code below, data item `PERSON-REC` is defined as a group containing another group and a numeric data item. `PERSON-DATA` is defined as an alphanumeric data item that renames `PERSON-REC`, treating the data bytes continued within it as character data.

```
01  PERSON-REC.
    05  PERSON-NAME.
```

```
          10  PERSON-NAME-LAST    PIC X(12).
          10  PERSON-NAME-FIRST   PIC X(16).
          10  PERSON-NAME-MID     PIC X.
      05  PERSON-ID               PIC 9(9) PACKED-DECIMAL.

  01  PERSON-DATA                 RENAMES PERSON-REC.
```

The second way to define a union type is by using the `REDEFINES` keyword. In the example code below, data item `VERS-NUM` is defined as a 2-byte binary integer containing a version number. A second data item `VERS-BYTES` is defined as a two-character alphanumeric variable. Since the second item is *redefined* over the first item, the two items share the same address in memory, and therefore share the same underlying data bytes. The first item interprets the two data bytes as a binary value, while the second item interprets the bytes as character values.

```
  01  VERS-INFO.
      05  VERS-NUM        PIC S9(4) COMP.
      05  VERS-BYTES      PIC X(2)
                          REDEFINES VERS-NUM.
```

Syntax and Example [edit]

In C and C++, the syntax is:

```
union <name>
{
    <datatype>  <1st variable name>;
    <datatype>  <2nd variable name>;
    .
    .
    .
    <datatype>  <nth variable name>;
} <union variable name>;
```

A structure can also be a member of a union, as the following example shows:

```
union name1
{
    struct name2
    {
        int     a;
        float   b;
        char    c;
    } svar;
    int     d;
} uvar;
```

This example defines a variable `uvar` as a union (tagged as `name1`), which contains two members, a structure (tagged as `name2`) named `svar` (which in turn contains three members), and an integer variable named `d`.

Unions may occur within structures and arrays, and vice versa:

```
struct
{
    int flags;
    char *name;
    int utype;
    union {
        int ival;
        float fval;
        char *sval;
    } u;
} symtab[NSYM];
```

The number ival is referred to as symtab[i].u.ival and the first character of string sval by either of *symtab[i].u.sval or symtab[i].u.sval[0].

Difference between Union and Structure [edit]

A union is a class all of whose data members are mapped to the same address within its object. The size of an object of a union is, therefore, the size of its largest data member.

In a structure, all of its data members are stored in contiguous memory locations. The size of an object of a struct is, therefore, the size of the sum of all its data members.

This gain in space efficiency, while valuable in certain circumstances, comes at a great cost of safety: the program logic must ensure that it only reads the field most recently written along all possible execution paths. The exception is when unions are used for type conversion: in this case, a certain field is written and the subsequently read field is deliberately different.

An example illustrating this point is:

```
                                    +-----+-----+
struct { int a; float b; }  gives   |  a  |  b  |
                                    +-----+-----+
                                       ^     ^
                                       |     |
                    memory location:  150   154
                                       |
                                       ˇ
                                    +-----+
union { int a; float b; }  gives    |  a  |
                                    |  b  |
                                    +-----+
```

Structures are used where an "object" is composed of other objects, like a point object consisting of two integers, those being the x and y coordinates:

```
typedef struct {
    int x;            // x and y are separate
    int y;
} tPoint;
```

Unions are typically used in situation where an object can be one of many things but only one at a time, such as a type-less storage system:

```
typedef enum { STR, INT } tType;
typedef struct {
    tType typ;          // typ is separate.
    union {
        int ival;       // ival and sval occupy same memory.
        char *sval;
    };
} tVal;
```

See also [edit]

- Tagged union
- UNION operator

External links [edit]

- boost::variant, a type-safe alternative to C++ unions
- MSDN: Classes,Structures & Unions, for examples and syntax
- differences, differences between union & structure
- Difference between struct and union in C++

v · t · e	**Data structures**	[show]

v · t · e **Data types** [show]

Categories: Data types | Composite data types | C (programming language)