Chapter 56: Unions

Section 56.1: Using unions to reinterpret values

Some C implementations permit code to write to one member of a union type then read from another in order to perform a sort of reinterpreting cast (parsing the new type as the bit representation of the old one).

It is important to note however, this is not permitted by the C standard current or past and will result in undefined behavior, none the less is is a very common extension offered by compilers (so check your compiler docs if you plan to do this).

One real life example of this technique is the "Fast Inverse Square Root" algorithm which relies on implementation details of IEEE 754 floating point numbers to perform an inverse square root more quickly than using floating point operations, this algorithm can be performed either through pointer casting (which is very dangerous and breaks the strict aliasing rule) or through a union (which is still undefined behavior but works in many compilers):

```
union floatToInt
{
    int32_t intMember;
    float floatMember; /* Float must be 32 bits IEEE 754 for this to work */
};

float inverseSquareRoot(float input)
{
    union floatToInt x;
    int32_t i;
    float f;
    x.floatMember = input;     /* Assign to the float member */
    i = x.intMember;           /* Read back from the integer member */
    i = 0x5f3759df - (i >> 1);
    x.intMember = i;           /* Assign to the integer member */
    f = x.floatMember;         /* Read back from the float member */
    f = f * (1.5f - input * 0.5f * f * f);
    return f * (1.5f - input * 0.5f * f * f);
}
```

This technique was widely used in computer graphics and games in the past due to its greater speed compared to using floating point operations, and is very much a compromise, losing some accuracy and being very non portable in exchange for speed.

Section 56.2: Writing to one union member and reading from another

The members of a union share the same space in memory. This means that writing to one member overwrites the data in all other members and that reading from one member results in the same data as reading from all other members. However, because union members can have differing types and sizes, the data that is read can be interpreted differently, see http://stackoverflow.com/documentation/c/1119/structs-and-unions/9399/using-unions-to-reinterpret-values

The simple example below demonstrates a union with two members, both of the same type. It shows that writing to member `m_1` results in the written value being read from member `m_2` and writing to member `m_2` results in the written value being read from member `m_1`.

```
#include <stdio.h>
```

```c
union my_union /* Define union */
{
    int m_1;
    int m_2;
};

int main (void)
{
    union my_union u;              /* Declare union */
    u.m_1 = 1;                     /* Write to m_1 */
    printf("u.m_2: %i\n", u.m_2); /* Read from m_2 */
    u.m_2 = 2;                     /* Write to m_2 */
    printf("u.m_1: %i\n", u.m_1); /* Read from m_1 */
    return 0;
}
```

Result

```
u.m_2: 1
u.m_1: 2
```

Section 56.3: Difference between struct and union

This illustrates that union members shares memory and that struct members does not share memory.

```c
#include <stdio.h>
#include <string.h>

union My_Union
{
  int variable_1;
  int variable_2;
};

struct My_Struct
{
  int variable_1;
  int variable_2;
};

int main (void)
{
  union My_Union u;
  struct My_Struct s;
  u.variable_1 = 1;
  u.variable_2 = 2;
  s.variable_1 = 1;
  s.variable_2 = 2;
  printf ("u.variable_1: %i\n", u.variable_1);
  printf ("u.variable_2: %i\n", u.variable_2);
  printf ("s.variable_1: %i\n", s.variable_1);
  printf ("s.variable_2: %i\n", s.variable_2);
  printf ("sizeof (union My_Union): %i\n", sizeof (union My_Union));
  printf ("sizeof (struct My_Struct): %i\n", sizeof (struct My_Struct));
  return 0;
}
```