A Tale of Two Unions

Ashwin Menon

December 15, 2024

Bjarne Stroustrup, in *The C++ Programming Language*, recommends avoiding the use of unions. This is of course good advice, mainly because unions are error prone due to the mixing of types that they entail. If you really do need to use unions (for performace or space reasons) then he advices wrapping them in classes. But does that increase the size of the assembly code generated? Let's investigate.

One of my projects was a *BASIC* interpreter, that needed to store variables values. Each value was a union that could hold an integer, a floating point type, and a string (more precisely, a pointer to a string descriptor). This project was written in *C*. Let's see if implementing this union using *C++* classes introduces code bloat. For this experiment, I will simplify the union by omitting the string descriptors, i.e. each variable value can be a float or an int.

First, the *C* code:

```c
enum ValueType {INT, REAL};

struct Value {
  enum ValueType type;
  union {
    int integer;
    float real;
  } val;
};

#define CONST_INT(C, VAL) \
  const struct Value C = { .type = INT, .val = { .integer = VAL }}
#define CONST_REAL(C, VAL) \
  const struct Value C = { .type = REAL, .val = { .real = VAL }}

CONST_INT(a, 42);
CONST_REAL(b, 1.5);

struct Value add_vars(const struct Value *a, const struct Value *b)
    {
  struct Value ret;
  ret.type = b->type;
  switch(b->type) {
    case INT:
      ret.val.integer = a->val.integer + b->val.integer;
      break;
    default:
```

```
      ret.val.real = a->val.real + b->val.real;
      break;
  }
  return ret;
}

void add_a(struct Value *v) {
  v->val.integer += a.val.integer;
}

void add_b(struct Value *v) {
  v->val.real += b.val.real;
}
```

The union itself is a standard tagged union. But we want to see what happens when we actually use the union. Since variables can be added in *BASIC*, I've added an `add_vars` function that does just that. To keep this simple, it assumes that the values passed in are of the same type (i.e. both `floats`'s or both `int`'s.).

BASIC has some built in constants (`PI`, for example). To simulate this, I've declared two constant values, an `int` and a `float`. I've also added two functions that take in a variable of the appropriate type and add one of these cosntants to it.

So what does this generate? I've used Matt Godbolt's excellent online tool to test the results. The code was compiled with the *-O2* optimization turned on. I've liberally commented the code.

A word about the x86_64 ABI. The compiler here is gcc, so the SystemV AMD64 ABI is used here, which is discussed in this Wikipedia article. In a nutshell, the first six integer parameters to a function are passed in *RDI*, *RSI*, *RDX*, *RCX*, *R8*, and *R9*, while *XMM0*, *XMM1*, *XMM2*, *XMM3*, *XMM4*, *XMM5*, *XMM6* and *XMM7* are used for the first floating point arguments. Additional arguments are passed on the stack. Integer return values upto 64 bits are returned in *RAX*.

```
; The add_vars function
; The parameter 'a' is passed in rdi
; The parameter 'b' is passed in rsi
add_vars:
        ; Store parameter b's type in eax
        mov     eax, DWORD PTR [rsi]
        ; Store parameter a's value in edx
        mov     edx, DWORD PTR [rdi+4]
        ; Store parameter b's value in ecx
        mov     ecx, DWORD PTR [rsi+4]
        ; Check if b's type is INT
        test    eax, eax
        jne     .L2
        ; If b is an INT, then add b's value to a's value using integer
        ; addition, and store the result in edx
        add     edx, ecx
        ; Move edx into the upper 32 bits of rdx, clearing the lower 32
```

```
        ; bits of rdx
        sal     rdx, 32
        ; Store b's type in the lower 32 bits of rdx
        or      rax, rdx
        ; At this point, rax contains the entire contents of the union.
        ; The upper 32 bits contain the value, and the lower 32 bits
        ; contain the type.
        ; Return the result in rax
        ret
; If b is a REAL, then add b's value to a's value using floating point
; addition
.L2:
        ; Store a's value in xmm0
        movd    xmm0, edx
        ; Store b's value in xmm1
        movd    xmm1, ecx
        ; Add a's value to b's value, storing the result in xmm0
        addss   xmm0, xmm1
        ; Move the result into edx
        movd    edx, xmm0
        ; Move edx into the upper 32 bits of rdx, clearing the lower 32
        ; bits of rdx
        sal     rdx, 32
        ; Store b's type in the lower 32 bits of rdx
        or      rax, rdx
        ; At this point, rax contains the entire contents of the union.
        ; The upper 32 bits contain the value, and the lower 32 bits
        ; contain the type.
        ; Return the result in rax
        ret

; The add_a function
add_a:
        ; The constant 'a' is 42, so just add 42 to 'a' and return
        add     DWORD PTR [rdi+4], 42
        ret

; The add_b function
add_b:
        ; Load the value of the constant 'b' into xmm0
        movss   xmm0, DWORD PTR .LC0[rip]
        ; Add the value of 'a' to 'b'
        ; Remember, the union 'a' is passed in rdi
        addss   xmm0, DWORD PTR [rdi+4]
        ; Store the result in the back into 'a' and return
        movss   DWORD PTR [rdi+4], xmm0
        ret

; The constant 'b'
b:
        .long   1
```

```
        .long   1069547520

; The constant 'a'
a:
        .long   0
        .long   42

; The constant 'b', again!
.LC0:
        .long   1069547520
```

No surprises here, except for the redundant location *LC0* storing the value of *b*; *add_b* could just have taken the value of *b* from the location *b*.

And now, the *C++* code:

```cpp
class Value {
  enum class ValueType {Integer, Real};

  enum ValueType type;
  union {
    int integer;
    float real;
  } val;

  public:
  // Constructors
  constexpr Value(int i): type{ValueType::Integer}, val{.integer{i}} { }
  constexpr Value(float f): type{ValueType::Real}, val{.real{f}} { }

  // Getting the value
  int integer() const { return val.integer; }
  float real() const { return val.real; }

  // + operator
  Value operator +(const Value& v) const {
    switch(v.type) {
      case ValueType::Integer:
        return Value(val.integer + v.integer());
      default:
        return Value(val.real + v.real());
      }
  }

  // += operator
  Value& operator +=(const Value& v) {
    switch(v.type) {
      case ValueType::Integer: val.integer += v.integer(); break;
      default: val.real += v.real(); break;
    }
    return *this;
  }
};

constexpr Value a{42};
constexpr Value b{1.5f};
```

```
Value add_vars(const Value& a, const Value& b) {
  return a+b;
}

void add_a(Value& v) {
  v += a;
}

void add_b(Value& v) {
  v += b;
}
```

What does this *C++* code generate? Take a look:

```
add_vars(Value const&, Value const&):
  mov     eax, DWORD PTR [rsi]
  mov     ecx, DWORD PTR [rdi+4]
  mov     edx, DWORD PTR [rsi+4]
  test    eax, eax
  jne     .L2
  add     edx, ecx
  sal     rdx, 32
  or      rax, rdx
  ret
.L2:
  movd    xmm0, edx
  movd    xmm1, ecx
  mov     eax, 1
  addss   xmm0, xmm1
  movd    edx, xmm0
  sal     rdx, 32
  or      rax, rdx
  ret
add_a(Value&):
  add     DWORD PTR [rdi+4], 42
  ret
add_b(Value&):
  movss   xmm0, DWORD PTR .LC0[rip]
  addss   xmm0, DWORD PTR [rdi+4]
  movss   DWORD PTR [rdi+4], xmm0
  ret
.LC0:
  .long   1069547520
```

The assembly generated from the *C++* source is virtually identical! So much for "*C++* bloat", eh?

Ok, so if the *C++* code generates identical assembly to *C* code, then what advantages has implementing the union in *C++* given me? Let's see.

Take a look at how constants are defined in the two regimes. In *C*, we do this:

```
CONST_INT(a, 42);
CONST_REAL(b, 1.5);
```

That is just ugly; opaque `#define`'s getting in the way of code clarity. In contrast, in *C++*, we do this:

```
constexpr Value a{42};
constexpr Value b{1.5f};
```

Pretty transparent, I think. We don't even have to specify the type of the constant being declared, as we had to in *C*; the compiler deduces the type, thanks to the two constructors we've put in for the *Value* class.

Next, adding two values. In *C*'s *add* function, we had to do the right thing depending on the union's type tag:

```
struct Value add(const struct Value *a, const struct Value *b) {
  struct Value ret;
  ret.type = b->type;
  switch(b->type) {
    case INT:
      ret.val.integer = a->val.integer + b->val.integer;
      break;
    default:
      ret.val.real = a->val.real + b->val.real;
      break;
  }
  return ret;
}
```

This is a lot of noise. Look at how we add two values in the *C++* version:

```
Value add(const Value& a, const Value& b) {
  return a+b;
}
```

Add means Add, and nothing else! All the noise is pushed into the implementation of the *Value* class, so that the user of the class doesn't have to bother with it. You could argue that the *add* function in *C* is doing the same thing, i.e. hiding implementation behind an API. But in *C++*, I don't even need the *add* function; I only added it here to show you the assembly generated by adding two values. In *C++*, I can simply use the + operator to add two values.

So in conclusion, a modern compiler will give you a tight implementation of your code, and by using *C++*, you can achive all the code clarity that using domain specific classes (in this case, the *Value* class) gives you.