Understanding C/C++ Strict Aliasing

or - Why won't the #$@##@^% compiler let me do what I need to do!

by Patrick Horgan

There's a lot of confusion about strict aliasing rules. The main source of people's confusion is that there are two different audiences that talk about aliasing, developers who use compilers, and compiler writers. In this document I'm going to try to clear it all up for you. The things that I'm going to cover are based on the aliasing rules in C89/90 (**6.3**), C98/99 (**6.5/7)** as well as in C++98 (**3.10/15**), and C++0x (**3.10/10**). To find the aliasing rules in any current version of the C or C++ standards, search for "may not be aliased", which will find a footnote that refers back up to the section on allowable forms of aliasing.

Developers get interested in aliasing when a compiler gives them a warning about type punning and strict aliasing rules and they try to understand what the warnings mean. They Google for the warning message, they find references to the section on aliasing in one of the C or C++ specs and think, "Yes, that's what I'm trying to do, alias." Then they study that section of the appropriate spec like they're studying arcane runes and try to divine the rules that will let them do the things that they're trying to do. They think that the aliasing rules are written to tell them how to do type punning. They couldn't be more wrong.

The compiler writers know what the strict aliasing rules are for. They are written to let compiler writers know when they can safely assume that a change made through one variable won't affect the value of another variable, and conversely when they have to assume that two variables *might* actually refer to the same spot in memory.

So this document is divided into two parts. First I'll talk about what strict aliasing is and why it exists, and then I'll talk about how to do the kinds of things developers need to do in ways that won't come in conflict with those rules.

Part the first. What is aliasing exactly?

Aliasing is when more than one lvalue refers to the same memory location (when you hear lvalue, think of things (variables) that can be on the left-hand side of assignments). As an example:

```
int anint;
int *intptr=&anint;
```

If you change the value of `*intptr`, the value referenced by `anint` also changes because `*intptr` aliases `anint`, it's just another name for the same thing. Another example is:

```
int anint;
void foo(int &i1, int &i2);
foo(anint,anint);
```

Within the body of `foo`, since we used `anint` for both arguments, the two references, `i1`, and `i2` alias, i.e. refer to the same location when called this way.

What's the problem?

Examine the following code:

```
int anint;
void foo(double *dblptr)
{
    anint=1;
    *dblptr=3.14159;
    bar(anint);
}
```

Looking at this, it looks safe to assume that the argument to `bar()` is a constant `1`. In the bad old days compiler writers had to make worst-case aliasing assumptions, to support lots of crazy wild west legacy code, and could not say that it *was* safe to assume the argument to `bar` was `1.  They` had to insert code to reload the value of `anint` for the call, because the intervening assignment through `dblptr` could have changed the value of `anint` if dblptr pointed to it. It's *possible* that the call to `foo` was `foo((double *)&anint)`.

That's the problem that strict aliasing is intended to fix. There was low hanging fruit for compiler optimizer writers to pick and they wanted programmers to follow the aliasing rules so that they could pluck those fruit. Aliasing, and the problems it leads to, have been there as long as C has existed. The difference lately, is that compiler writers are being strict about the rules and enforcing them when optimization is in effect. In their respective standards, C and C++ include lists of the things that can legitimately alias, (see the next section), and in all other cases, compiler writers are allowed to assume no interactions between lvalues. Anything not on the list can be assumed to not alias, and compiler writers are free to do optimizations that make that assumption. For anything on the list, aliasing could possibly occur and compiler writers have to assume that it does. When compiler writers follow these lists, and assume that *your* code follows the rules, it's called *strict-aliasing*. Under strict-aliasing, the compiler writer is free to optimize the function `foo` above because incompatible types, `double` and `int`, can't alias. That means that if you *do* call `foo` as `foo((double *)&anint)` something will go quickly wrong, but you get what you deserve.

So what can alias?

From C9899:201x *6.5 Expressions*:

7 An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

— a type compatible with the effective type of the object,

— a qualified version of a type compatible with the effective type of the object,

— a type that is the signed or unsigned type corresponding to the effective type of the object,

— a type that is the signed or unsigned type corresponding to a qualified version of the effective type of the object,

— an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or

— a character type.

These can be summarized as follows:

- Things that are compatible types or differ only by the addition of any combination of `signed`, `unsigned`, `const` or `volatile`. For most purposes compatible type just means the same type. If you want more details you can read the specs. (Example: If you get a pointer to `long`, and a pointer to `const unsigned long` they could point to the same thing.)

- An aggregate (`struct` or `class`) or `union` type can alias types contained inside them. (Example: If a function gets passed a pointer to an `int`, and a pointer to a `struct` or `union` containing an `int`, or possibly containing another `struct` or `union` containing an `int,` or containing...ad infinitum, it's possible that the `int*` points to an `int` contained inside the `struct` or `union` pointed at by the other pointer.)
- A character type. A `char*`, `signed char*`, or `unsigned char*` is specifically allowed by the specs to point to anything. That means it can alias anything in memory.
- For C++ only, a possibly CV (`const` and/or `volatile`) qualified base class type of a dynamic type can alias the child type. (Example: if `class dog` has `class animal` for a base class, pointers or references to `class dog` and `class animal` can alias.)

Of course references have all these same issues and pointers and references can alias. Any lvalue has to be assumed to possibly alias to another lvalue if these rules say that they can alias. It's just as likely to come up with values passed by reference or passed as pointer to values.

Part the second. How to do something the compiler doesn't like.

The following program swaps the halves of a 32 bit integer, and is typical of code you might use to handle data passed between a little-endian and big-endian machine. It also generates 6 warnings about breaking strict-aliasing rules. Many would dismiss them. The correct output of the program is:

00000020 00200000

but when optimization is turned on it's:

00000020 00000020

THAT's what the warning is trying to tell you, that the optimizer is going to do things that you don't like.

Broken Version

```
uint32_t
swaphalves(uint32_t a)
{
    uint32_t acopy=a;
    uint16_t *ptr=(uint16_t*)&acopy;// can't use static_cast<>, not legal.
                                    // you should be warned by that.
    uint16_t tmp=ptr[0];
    ptr[0]=ptr[1];
    ptr[1]=tmp;
    return acopy;
}

int main()
{
    uint32_t a;
    a=32;
    cout << hex << setfill('0') << setw(8) << a << endl;
    a=swaphalves(a);
    cout << setw(8) << a << endl;
}
```

So what goes wrong? Since a `uint16_t` can't alias a `uint32_t`, under the rules, it's ignored in considering what to do with `acopy`. Since it sees that nothing is done with `acopy` inside the `swaphalves` function, it just returns the original value of `a`. Here's the (annotated) x86 assembler generated by gcc 4.4.1 for `swaphalves`, let's see what went wrong:

```
_Z10swaphalvesj:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    8(%ebp), %eax   # get a in %eax
    movl    %eax, -8(%ebp)  # and store it in acopy
    leal    -8(%ebp), %eax  # now get eax pointing at acopy (ptr=&acopy)
    movl    %eax, -12(%ebp) # save that ptr at -12(%ebp)
    movl    -12(%ebp), %eax # get the ptr back in %eax
    movzwl  (%eax), %eax    # get 16 bits from ptr[0] in eax
    movw    %ax, -2(%ebp)   # store the 16 bits into tmp
    movl    -12(%ebp), %eax # get the ptr back in eax
    addl    $2, %eax        # bump up by two to get to ptr[1]
    movzwl  (%eax), %edx    # get that 16 bits into %edx
    movl    -12(%ebp), %eax # get ptr into eax
    movw    %dx, (%eax)     # store the 16 bits into ptr[1]
    movl    -12(%ebp), %eax # get the ptr again
    leal    2(%eax), %edx   # get the address of ptr[1] into edx
    movzwl  -2(%ebp), %eax  # get tmp into eax
    movw    %ax, (%edx)     # store into ptr[1]
    movl    -8(%ebp), %eax  # forget all that, return original a.
    leave
    ret
```

Scary, isn't it? Of course, if you *are* using gcc, you could use -fno-strict-aliasing to get the right output, but the generated code won't be as good. A better way to accomplish the same thing without the warnings or the incorrect output is to define `swaphalves` like this. N.B. this is supported in C99 and later C specs, as noted in this footnote to **6.5.2.3 Structure and union members:**

```
85)   If the member used to access the contents of a union object is not the
same as the member last used to store a value in the object, the appropriate
part of the object representation of the value is reinterpreted as an object
representation in the new type as described in 6.2.6 (a process sometimes
called ‘‘type punning’’). This might be a trap representation.
```

but your mileage may vary in C++, almost all compilers support it, but the spec doesn't allow it. Right after this discussion I'll have another solution with memcpy that may be slightly less efficient, (but probably not), and is supported by both C and C++):

Union version. Fixed for C but not guaranteed portable to C++.

```
uint32_t
swaphalves(uint32_t a)
{
    typedef union {
        uint32_t as32bit;
        uint16_t as16bit[2];
    } swapem;

    swapem s={a};
    uint16_t tmp;
    tmp=s.as16bit[0];
    s.as16bit[0]=s.as16bit[1];
    s.as16bit[1]=tmp;
    return s.as32bit;
}
```

The C++ compiler knows that members of a `union` fill the same memory, and this helps the compiler generate MUCH better code:

```
_Z10swaphalvesj:
    pushl   %ebp                # save the original value of ebp
    movl    %esp, %ebp          # point ebp at the stack frame
    movl    8(%ebp), %eax       # get a in eax
    popl    %ebp                # get the original ebp value back
    roll    $16, %eax           # swap the two halves of a and return it
    ret
```

So do it wrong, via strange casts and get incorrect code, or by turning off strict-aliasing get inefficient code, or do it right and get efficient code.

You can also accomplish the same thing by using memcpy with char* to move the data around for the swap, and it will probably be as efficient. *Wait*, you ask me, how can that be? The will be at least to calls to memcpy added to the mix! Well gcc and other modern compilers have smart optimizers and will, in many cases, (including this one), elide the calls to memcpy. That makes it the *most* portable, and as efficient as any other method. Here's how it would look:

memcpy version, compliant to C and C++ specs and efficient

```
uint32_t
swaphalves(uint32_t a)
{
    uint16_t as16bit[2],tmp;

    memcpy(as16bit, &a, sizeof(a));
    tmp = as16bit[0];
    as16bit[0] = as16bit[1];
    as16bit[1] = tmp;
    memcpy(&a, as16bit, sizeof(a));
    return a;
}
```

For the above code, a C compiler will generate code similar to the previous solution, but with the addition of two calls to memcpy (possibly optimized out). gcc generates code *identical* to the previous solution. You can imagine other variants that substitute reading and writing through a char pointer locally for the calls to memcpy.

Similar issues arrive from networking code where you don't know what type of packet you have until you examine it. unions are your friend here as well.

The restrict keyword

In C, but not in C++ you can promise the compiler that a pointer to something is not aliased with the *restrict* keyword. In a situation where the compiler would have to expect that things could alias, you can tell the compiler that you promise it will not be so. So in this:

```
void foo(int * restrict i1, int * restrict i2);
```

you're telling the compiler that you *promise* that `i1` and `i2` will never point at the same memory. You have to know well the implementation of `foo` and only pass into it things that will keep the promise that things accessed through `i1` and `i2` will never alias. The compiler believes you and may be able to do a better job of optimization. If you break the promise your mileage may vary (and by that I mean that you may cry). This is not available for C++.

If you have comments, corrections, suggestions for improvement, or examples, feel free to email me.

Thanks,

Patrick Horgan

patrick at dbp-consulting dot com

Particular thanks go to people who participated in the discussion of this document on the boost-users and gcc-help mailing lists. In particular I'd like to thank Václav Haisman, Thomas Heller who wrote

the memcpy version I use here and pointed out that it will generate exactly the same assembler, and Andrew Haley who pointed out a more portable way to define the union, and also pointed out that gcc will elide the calls to memcpy.