What is the strict aliasing rule?

Asked 16 years, 3 months ago Modified 11 months ago

Viewed 313k times

1007

When asking about common undefined behavior in C, people sometimes refer to the strict aliasing rule.
What are they talking about?

c++ c undefined-behavior strict-aliasing type-punning

Share Follow

edited Jun 9, 2021 at 18:24

NathanOliver

179k ● 29 ● 314 ● 427

asked Sep 19, 2008 at 1:30

Benoit

38.9k ● 24 ● 85 ● 117

11 May also want to see an article I wrote recently What is the Strict Aliasing Rule and Why do we care?. It covers a lot of material not covered here or in some areas a more modern approach. – Shafik Yaghmour Apr 22, 2018 at 4:06

Most strict aliasing violations are violations of eel.is/c++draft/basic.lval#11 in the C++ standard. (or the corresponding rule in the C standard) – Jan Schultke Sep 1, 2023 at 13:50

11 Answers

Sorted by: Highest score (default)

708

A typical situation where you encounter strict aliasing problems is when overlaying a struct (like a device/network msg) onto a buffer of the word size of your system (like a pointer to `uint32_t`s or `uint16_t`s). When you overlay a struct onto such a buffer, or a buffer onto such a struct through pointer casting you can easily violate strict aliasing rules.

So in this kind of setup, if I want to send a message to something I'd have to have two incompatible pointers pointing to the same chunk of memory. I might then naively code something like this:

```
typedef struct Msg
{
    unsigned int a;
    unsigned int b;
} Msg;

void SendWord(uint32_t);

int main(void)
{
    // Get a 32-bit buffer from the system
    uint32_t* buff = malloc(sizeof(Msg));

    // Alias that buffer through message
    Msg* msg = (Msg*)(buff);

    // Send a bunch of messages
    for (int i = 0; i < 10; ++i)
    {
        msg->a = i;
        msg->b = i+1;
        SendWord(buff[0]);
```

```
            SendWord(buff[1]);
        }
    }
```

The strict aliasing rule makes this setup illegal: dereferencing a pointer that aliases an object that is not of a compatible type or one of the other types allowed by C 2011 6.5 paragraph 7[1] is undefined behavior. Unfortunately, you can still code this way, *maybe* get some warnings, have it compile fine, only to have weird unexpected behavior when you run the code.

(GCC appears somewhat inconsistent in its ability to give aliasing warnings, sometimes giving us a friendly warning and sometimes not.)

To see why this behavior is undefined, we have to think about what the strict aliasing rule buys the compiler. Basically, with this rule, it doesn't have to think about inserting instructions to refresh the contents of `buff` every run of the loop. Instead, when optimizing, with some annoyingly unenforced assumptions about aliasing, it can omit those instructions, load `buff[0]` and `buff[1]` into CPU registers once before the loop is run, and speed up the body of the loop. Before strict aliasing was introduced, the compiler had to live in a state of paranoia that the contents of `buff` could change by any preceding memory stores. So to get an extra performance edge, and assuming most people don't type-pun pointers, the strict aliasing rule was introduced.

Keep in mind, if you think the example is contrived, this might even happen if you're passing a buffer to another function doing the sending for you, if instead you have.

```
void SendMessage(uint32_t* buff, size_t size32)
{
    for (int i = 0; i < size32; ++i)
    {
        SendWord(buff[i]);
    }
}
```

And rewrote our earlier loop to take advantage of this convenient function

```
for (int i = 0; i < 10; ++i)
{
    msg->a = i;
    msg->b = i+1;
    SendMessage(buff, 2);
}
```

The compiler may or may not be able to or smart enough to try to inline SendMessage and it may or may not decide to load or not load buff again. If `SendMessage` is part of another API that's compiled separately, it probably has instructions to load buff's contents. Then again, maybe you're in C++ and this is some templated header only implementation that the compiler thinks it can inline. Or maybe it's just something you wrote in your .c file for your own convenience. Anyway undefined behavior might still ensue. Even when we know some of what's happening under the hood, it's still a violation of the rule so no well defined behavior is guaranteed. So just by wrapping in a function that takes our word delimited buffer doesn't necessarily help.

So how do I get around this?

- Use a union. Most compilers support this without complaining about strict aliasing. This is allowed in C99 and explicitly allowed in C11.

  ```
  union {
      Msg msg;
      unsigned int asBuffer[sizeof(Msg)/sizeof(uns
  };
  ```

- You can disable strict aliasing in your compiler (f[no-]strict-aliasing in gcc))
- You can use `char*` for aliasing instead of your system's word. The rules allow an exception for `char*` (including `signed char` and `unsigned char`). It's always assumed that `char*` aliases other types. However this won't work the other way: there's no assumption that your struct aliases a buffer of chars.

Beginner beware

This is only one potential minefield when overlaying two types onto each other. You should also learn about endianness, word alignment, and how to deal with alignment issues through packing structs correctly.

Footnote

[1] The types that C 2011 6.5 7 allows an lvalue to access are:

- 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.

Share Follow

edited Feb 6, 2021 at 18:34

community wiki
36 revs, 21 users 73%
Doug T.

---

22 I am coming after the battle it seems.. may `unsigned char*` be used far `char*` instead ? I tend to use `unsigned char` rather than `char` as the underlying type for `byte` because my bytes are not signed and I don't want

the weirdness of signed behavior (notably wrt to overflow)
– Matthieu M. Nov 12, 2010 at 12:48

31 @Matthieu: Signedness makes no difference to alias rules, so using `unsigned char *` is okay. – Thomas Eding Jun 1, 2011 at 21:24

33 Isn't it undefined behaviour to read from an union member different from the last one written to?
– R. Martinho Fernandes Sep 6, 2011 at 14:41

30 Bollocks, this answer is *completely backwards*. The example it shows as illegal is actually legal, and the example it shows as legal is actually illegal. – R. Martinho Fernandes Sep 22, 2011 at 3:08

10 This example is unclear. It shows `unsigned int` being aliased as `uint32_t`. However it is implementation-defined whether `unsigned int` is compatible with `uint32_t`. If `uint32_t` is a typedef for `unsigned int` then the code actually has no strict aliasing violation, otherwise it does. I'd suggest editing it so that the two types are clearly incompatible. – M.M Jan 12, 2015 at 23:17

291

The best explanation I have found is by Mike Acton, Understanding Strict Aliasing. It's focused a little on PS3 development, but that's basically just GCC.

From the article:

> "Strict aliasing is an assumption, made by the C (or C++) compiler, that dereferencing pointers to objects of different types will never refer to the same memory location (i.e. alias each other.)"

So basically if you have an `int*` pointing to some memory containing an `int` and then you point a `float*` to that memory and use it as a `float` you break the rule. If your code does not respect this, then the compiler's optimizer will most likely break your code.

The exception to the rule is a `char*`, which is allowed to point to any type.

Share Follow

edited Oct 16, 2017 at 14:46
Palec
13.5k ● 8 ● 74 ● 142

answered Sep 19, 2008 at 1:38
Niall
5,121 ● 1 ● 22 ● 12

9 So what is the canonical way to legally use the same memory with variables of 2 different types? or does everyone just copy? – jiggunjer Jul 15, 2015 at 13:24

6 Mike Acton's page is flawed. The part of "Casting through a union (2)", at least, is downright wrong; the code he claims is legal is not. – davmac Sep 6, 2015 at 12:13

29 @davmac: The authors of C89 never intended that it should force programmers to jump through hoops. I find thoroughly bizarre the notion that a rule that exists for the sole purpose of optimization should be interpreted in such fashion as to require programmers to write code that redundantly copies data in the hopes that an optimizer will remove the redundant code. – supercat Jun 13, 2016 at 23:30

12 @curiousguy: False. Firstly, the original conceptual idea behind unions was that at any moment there's **only one** member object "active" in the given union object, while the

others simply don't exist. So, there are no "different objects at the same address" as you seem to believe. Secondly, aliasing violations everyone is talking about is about **accessing** one object as a different object, not about simply *having* two objects with the same address. As long as there is no type-punning **access**, there no problem. That was the original idea. Later, type-punning through unions was allowed. – AnT stands with Russia Nov 27, 2017 at 3:01

4 The exception is wider than `char *` --> Applies to any character type. – chux May 8, 2020 at 18:51

206

Note

This is excerpted from my "What is the Strict Aliasing Rule and Why do we care?" write-up.

What is strict aliasing?

In C and C++ aliasing has to do with what expression types we are allowed to access stored values through. In both C and C++ the standard specifies which expression types are allowed to alias which types. The compiler and optimizer are allowed to assume we follow the aliasing rules strictly, hence the term *strict aliasing rule*. If we attempt to access a value using a type not allowed it is classified as undefined behavior (**UB**). Once we have undefined behavior all bets are off, the results of our program are no longer reliable.

Unfortunately with strict aliasing violations, we will often obtain the results we expect, leaving the possibility the a future version of a compiler with a new optimization will break code we thought was valid. This is undesirable and it is a worthwhile goal to understand the strict aliasing rules and how to avoid violating them.

To understand more about why we care, we will discuss issues that come up when violating strict aliasing rules, type punning since common techniques used in type punning often violate strict aliasing rules and how to type pun correctly.

Preliminary examples

Let's look at some examples, then we can talk about exactly what the standard(s) say, examine some further examples and then see how to avoid strict aliasing and catch violations we missed. Here is an example that should not be surprising (live example):

```
int x = 10;
int *ip = &x;

std::cout << *ip << "\n";
*ip = 12;
std::cout << x << "\n";
```

We have a *int** pointing to memory occupied by an *int* and this is a valid aliasing. The optimizer must assume that assignments through **ip** could update the value occupied by **x**.

The next example shows aliasing that leads to undefined behavior (live example):

```
int foo( float *f, int *i ) {
    *i = 1;
    *f = 0.f;

    return *i;
}

int main() {
    int x = 0;

    std::cout << x << "\n";   // Expect 0
    x = foo(reinterpret_cast<float*>(&x), &x);
    std::cout << x << "\n";   // Expect 0?
}
```

In the function **foo** we take an *int** and a *float**, in this example we call **foo** and set both parameters to point to the same memory location which in this example contains an *int*. Note, the reinterpret_cast is telling the compiler to treat the expression as if it had the type specified by its template parameter. In this case we are telling it to treat the expression **&x** as if it had type *float**. We may naively expect the result of the second **cout** to be **0** but with optimization enabled using **-O2** both gcc and clang produce the following result:

```
0
1
```

Which may not be expected but is perfectly valid since we have invoked undefined behavior. A *float* can not validly alias an *int* object. Therefore the optimizer can assume the *constant 1* stored when dereferencing **i** will be the return value since a store through **f** could not validly affect an *int* object. Plugging the code in Compiler Explorer shows this is exactly what is happening(live example):

```
foo(float*, int*): # @foo(float*, int*)
mov dword ptr [rsi], 1
mov dword ptr [rdi], 0
mov eax, 1
ret
```

The optimizer using Type-Based Alias Analysis (TBAA) assumes **1** will be returned and directly moves the constant value into register **eax** which carries the return value. TBAA uses the languages rules about what types are allowed to alias to optimize loads and stores. In this case TBAA knows that a *float* can not alias an *int* and optimizes away the load of **i**.

Now, to the Rule-Book

What exactly does the standard say we are allowed and not allowed to do? The standard language is not straightforward, so for each item I will try to provide code examples that demonstrates the meaning.

What does the C11 standard say?

The **C11** standard says the following in section *6.5 Expressions paragraph 7*:

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types:88) — a type compatible with the effective type of the object,

```
int x = 1;
int *p = &x;
printf("%d\n", *p); // *p gives us an lvalue expressio
compatible with int
```

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

```
int x = 1;
const int *p = &x;
printf("%d\n", *p); // *p gives us an lvalue expressio
is compatible with int
```

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

```
int x = 1;
unsigned int *p = (unsigned int*)&x;
printf("%u\n", *p ); // *p gives us an lvalue expressi
which corresponds to
                     // the effective type of the obje
```

gcc/clang has an extension and also that allows assigning *unsigned int** to *int** even though they are not compatible types.

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

```
int x = 1;
const unsigned int *p = (const unsigned int*)&x;
printf("%u\n", *p ); // *p gives us an lvalue expressi
int which is a unsigned type
                     // that corresponds with to a qua
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

```
struct foo {
    int x;
};

void foobar( struct foo *fp, int *ip );  // struct foo
includes int among its members so it
                                         // can alias

foo f;
foobar( &f, &f.x );
```

> — a character type.

```
int x = 65;
char *p = (char *)&x;
printf("%c\n", *p );  // *p gives us an lvalue express
```

```
a character type.
                              // The results are not portable
```

What the C++17 Draft Standard says

The C++17 draft standard in section *[basic.lval] paragraph 11* says:

> If a program attempts to access the stored value of an object through a glvalue of other than one of the following types the behavior is undefined:[63]

> (11.1) — the dynamic type of the object,

```
void *p = malloc( sizeof(int) ); // We have allocated
the lifetime of an object
int *ip = new (p) int{0};        // Placement new chan
the object to int
std::cout << *ip << "\n";        // *ip gives us a glv
int which matches the dynamic type
                                 // of the allocated o
```

> (11.2) — a cv-qualified version of the dynamic type of the object,

```
int x = 1;
const int *cip = &x;
std::cout << *cip << "\n";  // *cip gives us a glvalue
```

```
int which is a cv-qualified
                                        // version of the dynamic
```

> (11.3) — a type similar (as defined in 7.5) to the dynamic type of the object,

> (11.4) — a type that is the signed or unsigned type corresponding to the dynamic type of the object,

```
// Both si and ui are signed or unsigned types corresp
dynamic types
// We can see from this godbolt(https://godbolt.org/g/
assumes aliasing.
signed int foo( signed int &si, unsigned int &ui ) {
    si = 1;
    ui = 2;

    return si;
}
```

> (11.5) — a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,

```
signed int foo( const signed int &si1, int &si2); // H
assumes aliasing
```

> (11.6) — an aggregate or union type that includes one of the aforementioned types among

> its elements or nonstatic data members (including, recursively, an element or non-static data member of a subaggregate or contained union),

```
struct foo {
    int x;
};

// Compiler Explorer example(https://godbolt.org/g/z2w
assumption
int foobar( foo &fp, int &ip ) {
    fp.x = 1;
    ip = 2;

    return fp.x;
}

foo f;
foobar( f, f.x );
```

> (11.7) — a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,

```
struct foo { int x; };

struct bar : public foo {};

int foobar( foo &f, bar &b ) {
    f.x = 1;
    b.x = 2;

    return f.x;
}
```

> (11.8) — a char, unsigned char, or std::byte type.

```
int foo( std::byte &b, uint32_t &ui ) {
    b = static_cast<std::byte>('a');
    ui = 0xFFFFFFFF;

    return std::to_integer<int>( b );  // b gives us a
type std::byte which can alias
                                       // an object of
}
```

Worth noting *signed char* is not included in the list above, this is a notable difference from *C* which says *a character type*.

What is Type Punning

We have gotten to this point and we may be wondering, why would we want to alias for? The answer typically is to *type pun*, often the methods used violate strict aliasing rules.

Sometimes we want to circumvent the type system and interpret an object as a different type. This is called *type punning*, to reinterpret a segment of memory as another type. *Type punning* is useful for tasks that want access to the underlying representation of an object to view, transport or manipulate. Typical areas we find type punning being used are compilers, serialization, networking code, etc…

Traditionally this has been accomplished by taking the address of the object, casting it to a pointer of the type we want to reinterpret it as and then accessing the value, or in other words by aliasing. For example:

```
int x = 1;

// In C
float *fp = (float*)&x;  // Not a valid aliasing

// In C++
float *fp = reinterpret_cast<float*>(&x);  // Not a va

printf( "%f\n", *fp );
```

As we have seen earlier this is not a valid aliasing, so we are invoking undefined behavior. But traditionally compilers did not take advantage of strict aliasing rules and this type of code usually just worked, developers have unfortunately gotten used to doing things this way. A common alternate method for type punning is through unions, which is valid in C but *undefined behavior* in C++ (see live example):

```
union u1
{
    int n;
    float f;
};

union u1 u;
u.f = 1.0f;

printf( "%d\n", u.n );  // UB in C++ n is not the acti
```

This is not valid in C++ and some consider the purpose of unions to be solely for implementing variant types and feel using unions for type punning is an abuse.

How do we Type Pun correctly?

The standard method for *type punning* in both C and C++ is **memcpy**. This may seem a little heavy handed but the optimizer should recognize the use of **memcpy** for *type punning* and optimize it away and generate a register to register move. For example if we know *int64_t* is the same size as *double*:

```
static_assert( sizeof( double ) == sizeof( int64_t ) )
require a message
```

we can use **memcpy**:

```
void func1( double d ) {
    std::int64_t n;
    std::memcpy(&n, &d, sizeof d);
    //...
```

At a sufficient optimization level any decent modern compiler generates identical code to the previously mentioned **reinterpret_cast** method or *union* method for *type punning*. Examining the generated code we see it uses just register mov (live Compiler Explorer Example).

C++20 and bit_cast

In C++20 we may gain **bit_cast** (implementation available in link from proposal) which gives a simple and safe way to type-pun as well as being usable in a constexpr context.

The following is an example of how to use **bit_cast** to type pun a *unsigned int* to *float*, (see it live):

```
std::cout << bit_cast<float>(0x447a0000) << "\n"; //as
sizeof(unsigned int)
```

In the case where *To* and *From* types don't have the same size, it requires us to use an intermediate struct15. We will use a struct containing a **sizeof(unsigned int)** character array (*assumes 4 byte unsigned int*) to be the *From* type and *unsigned int* as the *To* type.:

```
struct uint_chars {
    unsigned char arr[sizeof( unsigned int )] = {};  /
int ) == 4
};

// Assume len is a multiple of 4
int bar( unsigned char *p, size_t len ) {
    int result = 0;

    for( size_t index = 0; index < len; index += sizeo
        uint_chars f;
        std::memcpy( f.arr, &p[index], sizeof(unsigned
        unsigned int result = bit_cast<unsigned int>(f

        result += foo( result );
    }

    return result;
}
```

It is unfortunate that we need this intermediate type but that is the current constraint of **bit_cast**.

Catching Strict Aliasing Violations

We don't have a lot of good tools for catching strict aliasing in C++, the tools we have will catch some cases of strict aliasing violations and some cases of misaligned loads and stores.

gcc using the flag **-fstrict-aliasing** and **-Wstrict-aliasing** can catch some cases although not without false positives/negatives. For example the following cases will generate a warning in gcc (see it live):

```
int a = 1;
short j;
float f = 1.f; // Originally not initialized but tis-k
               // it was being accessed w/ an indeterm

printf("%i\n", j = *(reinterpret_cast<short*>(&a)));
printf("%i\n", j = *(reinterpret_cast<int*>(&f)));
```

although it will not catch this additional case (see it live):

```
int *p;

p = &a;
printf("%i\n", j = *(reinterpret_cast<short*>(p)));
```

Although clang allows these flags it apparently does not actually implement the warnings.

Another tool we have available to us is ASan which can catch misaligned loads and stores. Although these are not directly strict aliasing violations they are a common result of strict aliasing violations. For example the following cases will generate runtime errors when built with clang using **-fsanitize=address**

```
int *x = new int[2];               // 8 bytes: [0,7].
int *u = (int*)((char*)x + 6);     // regardless of al
not be an aligned address
*u = 1;                            // Access to range
printf( "%d\n", *u );              // Access to range
```

The last tool I will recommend is C++ specific and not strictly a tool but a coding practice, don't allow C-style casts. Both gcc and clang will produce a diagnostic for C-style casts using **-Wold-style-cast**. This will force any undefined type puns to use reinterpret_cast, in general reinterpret_cast should be a flag for closer code review. It is also easier to search your code base for reinterpret_cast to perform an audit.

For C we have all the tools already covered and we also have tis-interpreter, a static analyzer that exhaustively analyzes a program for a large subset of the C language. Given a C version of the earlier example where using **-fstrict-aliasing** misses one case (see it live)

```
int a = 1;
short j;
float f = 1.0;

printf("%i\n", j = *((short*)&a));
printf("%i\n", j = *((int*)&f));

int *p;

p = &a;
printf("%i\n", j = *((short*)p));
```

tis-interpeter is able to catch all three, the following example invokes tis-kernel as tis-interpreter (output is edited for brevity):

```
./bin/tis-kernel -sa example1.c
...
example1.c:9:[sa] warning: The pointer (short *)(& a)
violates strict aliasing
              rules by accessing a cell with effective
...

example1.c:10:[sa] warning: The pointer (int *)(& f) h
violates strict aliasing rules by
              accessing a cell with effective type flo
              Callstack: main
...

example1.c:15:[sa] warning: The pointer (short *)p has
violates strict aliasing rules by
              accessing a cell with effective type int
```

Finally there is TySan which is currently in development. This sanitizer adds type checking information in a shadow memory segment and checks accesses to see if they violate aliasing rules. The tool potentially should be able to catch all aliasing violations but may have a large run-time overhead.

Share Follow   edited Jan 22 at 19:37

Oliver Tušla
148 ● 4 ● 12

answered Jul 8, 2018 at 2:07
Shafik Yaghmour
158k ● 42 ● 461 ● 765

Comments are not for extended discussion; this conversation has been moved to chat. – Bhargav Rao Aug 5, 2018 at 21:51

Fantastic article. May I recommend in the first section here (and on github) include one simple sentence for those who have not yet learned what kind of "aliasing" the term even refers to. Suggestion: "Aliasing is when two or more values, pointers or references in your code (potentially or actually) refer to the same underlying memory." – Louis Semprini Sep 18 at 10:23

153

This is the strict aliasing rule, found in section 3.10 of the **C++03** standard (other answers provide good explanation, but none provided the rule itself):

> If a program attempts to access the stored value of an object through an lvalue of other than one of the following types the behavior is undefined:
>
> - the dynamic type of the object,
> - a cv-qualified version of the dynamic type of the object,
> - a type that is the signed or unsigned type corresponding to the dynamic type of the

> object,
>
> - a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic 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),
> - a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
> - a `char` or `unsigned char` type.

C++11 and **C++14** wording (changes emphasized):

> If a program attempts to access the stored value of an object through a *glvalue* of other than one of the following types the behavior is undefined:
>
> - the dynamic type of the object,
> - a cv-qualified version of the dynamic type of the object,
> - *a type similar (as defined in 4.4) to the dynamic type of the object,*
> - a type that is the signed or unsigned type corresponding to the dynamic type of the object,

> - a type that is the signed or unsigned type corresponding to a cv-qualified version of the dynamic type of the object,
> - an aggregate or union type that includes one of the aforementioned types among its *elements or non-static data members* (including, recursively, an *element or non-static data member* of a subaggregate or contained union),
> - a type that is a (possibly cv-qualified) base class type of the dynamic type of the object,
> - a `char` or `unsigned char` type.

Two changes were small: *glvalue* instead of *lvalue*, and clarification of the aggregate/union case.

The third change makes a stronger guarantee (relaxes the strong aliasing rule): The new concept of *similar types* that are now safe to alias.

Also the **C** wording (C99; ISO/IEC 9899:1999 6.5/7; the exact same wording is used in ISO/IEC 9899:2011 §6.5 ¶7):

> An object shall have its stored value accessed only by an lvalue expression that has one of the following types [73) or 88)]:

> - 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.
>
> 73) or 88) The intent of this list is to specify those circumstances in which an object may or may not be aliased.

Share Follow

edited Jun 20, 2020 at 9:12
Community Bot
1 ● 1

answered Aug 10, 2011 at 4:43
Ben Voigt
283k ● 44 ● 435 ● 745

1 Look at the C89 Rationale cs.technion.ac.il/users/yechiel/CS/C++draft/rationale.pdf

section 3.3 which talks about it. – phorgan1 Jan 5, 2012 at 1:44

2 If one has an lvalue of a structure type, takes the address of a member, and passes that to a function that uses it as a pointer to the member type, would that be regarded as accessing an object of the member type (legal), or an object of the structure type (forbidden)? A *lot* of code assumes it's legal to access structures in such fashion, and I think a lot of people would squawk at a rule which was understood as forbidding such actions, but it's unclear what the exact rules are. Further, unions and structures are treated the same, but sensible rules for each should be different. – supercat Nov 27, 2015 at 19:45

3 @supercat: The way the rule for structures is worded, the actual access is always to the primitive type. Then access via a reference to the primitive type is legal because the types match, and access via a reference to the containing structure type is legal because it's specially permitted. – Ben Voigt Nov 27, 2015 at 20:27

1 @BenVoigt: Under that interpretation, if `S1` and `S2` are structures with `int x;` as their first field, and which require nothing coarser than `int` alignment, then given `void blah(S1 *p1, S2, *p2`);` a compiler would not be allowed to make any assumptions about aliasing between `p1->x` and `p2->x`. because they could both identify storage of type `int`. I don't think that's what was intended. – supercat Nov 27, 2015 at 20:39

2 @BenVoigt: I don't think the common initial sequence works unless accesses are done via the union. See goo.gl/HGOyoK to see what gcc is doing. If accessing an lvalue of union type via an lvalue of a member type (not using union-member-access operator) was legal, then `wow(&u->s1,&u->s2)` would need to be legal even when a pointer is used to modify `u`, and that would negate most optimizations that the

aliasing rule was designed to facilitate. – supercat Nov 28, 2015 at 17:50

51

Strict aliasing doesn't refer only to pointers, it affects references as well, I wrote a paper about it for the boost developer wiki and it was so well received that I turned it into a page on my consulting web site. It explains completely what it is, why it confuses people so much and what to do about it. [Strict Aliasing White Paper](). In particular it explains why unions are risky behavior for C++, and why using memcpy is the only fix portable across both C and C++. Hope this is helpful.

Share Follow

answered Jun 19, 2011 at 23:46

phorgan1
1,744 ● 18 ● 18

4 The "Another Broken version, referencing a twice" section of the paper makes no sense. Even if there were a sequence point, it would not give the right result. Perhaps you meant to use shift operators instead of shift assignments? But then the code is well-defined and does the right thing. – Yakov Galka Nov 9, 2014 at 18:36

7 Good paper. My take: (1) this aliasing-'problem' is an over-reaction to bad programming - trying to protect the bad programmer from his/her bad habits. If the programmer has good habits then this aliasing is just a nuisance and the checks can safely be turned off. (2) Compiler-side optimization should only be done in well-known cases and should when in doubt strictly follow the source-code; forcing the programmer to write code to cater for the compiler's

idiosyncrasies is, simply put, wrong. Even worse to make it part of the standard. – slashmais Feb 2, 2015 at 8:29

5 @slashmais (1) "*is an over-reaction to bad programming*" Nonsense. It is a rejection of the bad habits. *You do that? You pay the price: no guarantee for you!* (2) Well known cases? Which ones? The strict aliasing rule should be "well known"! – curiousguy Aug 16, 2015 at 1:45

6 @curiousguy: Having cleared up a few points of confusion, it is clear that the C language with the aliasing rules makes it impossible for programs to implement type-agnostic memory pools. Some kinds of program can get by with malloc/free, but others need memory management logic better tailored to the tasks at hand. I wonder why the C89 rationale used such a crummy example of the reason for the aliasing rule, since their example makes it seem like the rule won't pose any major difficulty in performing any reasonable task. – supercat Nov 21, 2015 at 8:53

6 @curiousguy, most compiler suites out there are including -fstrict-aliasing as default on -O3 and this hidden contract is forced on the users who've never heard of the TBAA and wrote code like how a system programmer might. I don't mean to sound disingenuous to system programmers, but this kind of optimization should be left outside of default opt of -O3 and should be an opt-in optimization for those that know what TBAA is. It is not fun looking at compiler 'bug' that turns out to be user code violating TBAA, especially tracking down the source level violation in user code. – kchoi Jul 14, 2016 at 16:09

37

As addendum to what Doug T. already wrote, here is a simple test case which probably triggers it with gcc :

check.c

```c
#include <stdio.h>

void check(short *h,long *k)
{
    *h=5;
    *k=6;
    if (*h == 5)
        printf("strict aliasing problem\n");
}

int main(void)
{
    long      k[1];
    check((short *)k,k);
    return 0;
}
```

Compile with `gcc -O2 -o check check.c` . Usually (with most gcc versions I tried) this outputs "strict aliasing problem", because the compiler assumes that "h" cannot be the same address as "k" in the "check" function. Because of that the compiler optimizes the `if (*h == 5)` away and always calls the printf.

For those who are interested here is the x64 assembler code, produced by gcc 4.6.3, running on ubuntu 12.04.2 for x64:

```
movw    $5, (%rdi)
movq    $6, (%rsi)
movl    $.LC0, %edi
jmp puts
```

So the if condition is completely gone from the assembler code.

Share Follow

answered May 14, 2013 at 2:37

Ingo Blackman
990 ● 8 ● 13

if you add a second short * j to check() and use it (*j = 7) then optimization disapear since ggc does not not if h and j are not actualy point to same value. yes optimisation is really smart. – philippe lhardy Dec 30, 2013 at 20:30

3 To make things more fun, use pointers to types which aren't compatible but have the same size and representation (on some systems that's true of e.g. `long long*` and `int64_t` *). One might expect that a sane compiler should recognize that a `long long*` and `int64_t*` could access the same storage if they're stored identically, but such treatment is no longer fashionable. – supercat Mar 22, 2017 at 19:54

2 Grr... x64 is a Microsoft convention. Use amd64 or x86_64 instead. – S.S. Anne Jul 30, 2019 at 20:39

25

According to the C89 rationale, the authors of the Standard did not want to require that compilers given code like:

```
int x;
int test(double *p)
{
  x=5;
  *p = 1.0;
  return x;
}
```

should be required to reload the value of `x` between the assignment and return statement so as to allow for the possibility that `p` might point to `x`, and the assignment to `*p` might consequently alter the value of `x`. The notion that a compiler should be entitled to presume that there won't be aliasing *in situations like the above* was non-controversial.

Unfortunately, the authors of the C89 wrote their rule in a way that, if read literally, would make even the following function invoke Undefined Behavior:

```
void test(void)
{
  struct S {int x;} s;
  s.x = 1;
}
```

because it uses an lvalue of type `int` to access an object of type `struct S`, and `int` is not among the types that may be used accessing a `struct S`. Because it would be absurd to treat all use of non-character-type members of structs and unions as Undefined Behavior, almost everyone recognizes that there are at least some circumstances where an lvalue of one type may be used to access an object of another type. Unfortunately, the C Standards Committee has failed to define what those circumstances are.

Much of the problem is a result of Defect Report #028, which asked about the behavior of a program like:

```
int test(int *ip, double *dp)
{
  *ip = 1;
  *dp = 1.23;
  return *ip;
}
int test2(void)
{
  union U { int i; double d; } u;
  return test(&u.i, &u.d);
}
```

Defect Report #28 states that the program invokes Undefined Behavior because the action of writing a union member of type "double" and reading one of type "int" invokes Implementation-Defined behavior. Such reasoning is nonsensical, but forms the basis for the Effective Type rules which needlessly complicate the language while doing nothing to address the original problem.

The best way to resolve the original problem would probably be to treat the footnote about the purpose of the rule as though it were normative, and made the rule unenforceable except in cases which actually involve conflicting accesses using aliases. Given something like:

```
 void inc_int(int *p) { *p = 3; }
 int test(void)
 {
   int *p;
   struct S { int x; } s;
   s.x = 1;
   p = &s.x;
   inc_int(p);
```

```
    return s.x;
  }
```

There's no conflict within `inc_int` because all accesses to the storage accessed through `*p` are done with an lvalue of type `int`, and there's no conflict in `test` because `p` is visibly derived from a `struct S`, and by the next time `s` is used, all accesses to that storage that will ever be made through `p` will have already happened.

If the code were changed slightly...

```
void inc_int(int *p) { *p = 3; }
int test(void)
{
  int *p;
  struct S { int x; } s;
  p = &s.x;
  s.x = 1;  //  !!*!!
  *p += 1;
  return s.x;
}
```

Here, there is an aliasing conflict between `p` and the access to `s.x` on the marked line because at that point in execution another reference exists *that will be used to access the same storage*.

Had Defect Report 028 said the original example invoked UB because of the overlap between the creation and use of the two pointers, that would have made things a lot more clear without having to add "Effective Types" or other such complexity.

Share Follow

edited Mar 10, 2018 at 0:42

answered Apr 26, 2017 at 22:42

supercat
80.8k ● 9 ● 174 ● 220

Well put, it would be interesting to read a proposal of sorts that was more or less "what the standards committee could have done" that achieved their goals without introducing as much complexity. – jrh Jul 31, 2018 at 13:40

1 @jrh: I think it would be pretty simple. Recognize that 1. For aliasing to occur during a particular execution of a function or loop, two different pointers or lvalues must be used *during that execution* to address the same storage in conflicting fashon; 2. Recognize that in contexts where one pointer or lvalue is freshly visibly derived from another, an access to the second is an access to the first; 3. Recognize that the rule is not intended to apply in cases that don't actually involve aliasing. – supercat Jul 31, 2018 at 14:47

1 The exact circumstances where a compiler recognizes a freshly-derived lvalue may be a Quality-of-Implementation issue, but any remotely-decent compiler should be able to recognize forms that gcc and clang deliberately ignore. – supercat Jul 31, 2018 at 14:49

The `s.x = 1;` example is not UB, because a struct type is compatible with the type of its first field. The object `s` stores both an `int` AND a `struct S`. – yyny Mar 6 at 9:36

@yyny: Both examples contain `s.x = 1;`, so I'm not clear which one you're talking about. If `p1` and `pp2` are pointers to different structure types that start with an `int[5];`, gcc won't recognize the possibility that a write to `*(pp2->x)` might affect the value of `p1->x[i]`, even though both are lvalues of type `int`. See godbolt.org/z/3xdqKjsMe for an example. – supercat Mar 6 at 16:09

22

Type punning via pointer casts (as opposed to using a union) is a major example of breaking strict aliasing.

Share Follow

edited Jul 6, 2014 at 18:31

answered Sep 19, 2008 at 1:38

C. K. Young
223k ● 47 ● 390 ● 443

1 See my answer here for the relevant quotes, especially the footnotes but type punning through unions has always been allowed in C although it was poorly worded at first. You my want to clarify your answer. – Shafik Yaghmour Jul 6, 2014 at 18:18

1 @ShafikYaghmour: C89 clearly allowed implementers to select the cases in which they would or would not usefully recognize type punning through unions. An implementation could, for example, specify that for a write to one type followed by a read of another to be recognized as type punning, if the programmer did either of the following *between the write and the read*: (1) evaluate an lvalue containing the union type [taking the address of a member would qualify, if done at the right point in the sequence]; (2) convert a pointer to one type into a pointer to the other, and access via that ptr. – supercat Mar 22, 2017 at 19:42

1 @ShafikYaghmour: An implementation could also specify e.g. that type punning between integer and floating-point values would only work reliably if code executed an `fpsync()`

directive between writing as fp and reading as int or vice versa [on implementations with separate integer and FPU pipelines and caches, such a directive might be expensive, but not as costly as having the compiler perform such synchronization on every union access]. Or an implementation could specify that the resulting value will never be usable except in circumstances using Common Initial Sequences. – supercat Mar 22, 2017 at 19:48

1 @ShafikYaghmour: Under C89, implementations *could* forbid most forms of type punning, including via unions, but the equivalence between pointers to unions and pointers to their members implied that type punning was allowed in implementations that didn't *expressly* forbid it. – supercat Mar 22, 2017 at 19:52

17

After reading many of the answers, I feel the need to add something:

Strict aliasing (which I'll describe in a bit) **is important because**:

1. Memory access can be expensive (performance wise), which is why **data is manipulated in CPU registers** before being written back to the physical memory.

2. If data in two different CPU registers will be written to the same memory space, **we can't predict which data will "survive"** when we code in C.

   In assembly, where we code the loading and unloading of CPU registers manually, we will know which data remains intact. But C (thankfully) abstracts this detail away.

Since two pointers can point to the same location in the memory, this could result in **complex code that handles possible collisions**.

This extra code is slow and **hurts performance** since it performs extra memory read / write operations which are both slower and (possibly) unnecessary.

The **Strict aliasing rule allows us to avoid redundant machine code** in cases in which it *should be* safe to assume that two pointers don't point to the same memory block (see also the `restrict` keyword).

The Strict aliasing states it's safe to assume that pointers to different types point to different locations in the memory.

If a compiler notices that two pointers point to different types (for example, an `int *` and a `float *`), it will assume the memory address is different and it **will not** protect against memory address collisions, resulting in faster machine code.

For example:

Lets assume the following function:

```
void merge_two_ints(int *a, int *b) {
  *b += *a;
```

```
    *a += *b;
}
```

In order to handle the case in which `a == b` (both pointers point to the same memory), we need to order and test the way we load data from the memory to the CPU registers, so the code might end up like this:

1. load `a` and `b` from memory.
2. add `a` to `b`.
3. **save** `b` and **reload** `a`.

   (save from CPU register to the memory and load from the memory to the CPU register).
4. add `b` to `a`.
5. save `a` (from the CPU register) to the memory.

Step 3 is very slow because it needs to access the physical memory. However, it's required to protect against instances where `a` and `b` point to the same memory address.

Strict aliasing would allow us to prevent this by telling the compiler that these memory addresses are distinctly different (which, in this case, will allow even further optimization which can't be performed if the pointers share a memory address).

1. This can be told to the compiler in two ways, by using different types to point to. i.e.:

```
void merge_two_numbers(int *a, long *b) {...}
```

2. Using the `restrict` keyword. i.e.:

```
void merge_two_ints(int * restrict a, int * restri
```

Now, by satisfying the Strict Aliasing rule, step 3 can be avoided and the code will run significantly faster.

In fact, by adding the `restrict` keyword, the whole function could be optimized to:

1. load `a` and `b` from memory.
2. add `a` to `b`.
3. save result both to `a` and to `b`.

This optimization couldn't have been done before, because of the possible collision (where `a` and `b` would be tripled instead of doubled).

Share Follow

edited Jan 16, 2018 at 14:11

answered Dec 24, 2017 at 12:04

Myst
19.2k ● 3 ● 51 ● 68

with restrict keyword, at step 3, shouldn't it be save result to 'b' only? It sounds as if the result of the summation will be stored in 'a' as well. Does it 'b' need to be reloaded again?
– NeilB Jan 16, 2018 at 14:08

1 @NeilB - Yap you're right. We're only saving `b` (not reloading it) and reloading `a`. I hope it's clearer now. – Myst Jan 16, 2018 at 14:13

---

1 Type-based aliasing may have offered some benefits prior to `restrict`, but I would think that the latter would in most circumstances be more effective, and loosening some constraints on `register` would allow it to fill in some of the cases where `restrict` wouldn't help. I'm not sure it was ever "important" to treat the Standard as fully describing all cases where programmers should expect compilers to recognize evidence of aliasing, rather than merely describing places where compilers must presume aliasing *even when no particular evidence of it exists*. – supercat Jan 23, 2018 at 18:33

---

1 Note that although loading from main RAM is very slow (and can stall the CPU core for a long time if following operations depend on the result), loading from L1 cache is pretty fast, and so is writing to a cache line that was recently writing to by the same core. So all but the first read or write to an address will usually be reasonably fast: the difference between reg/mem addr access is smaller than the difference between cached/uncached mem addr. – curiousguy Oct 24, 2019 at 20:23

@curiousguy - although you're correct, "fast" in this case is relative. The L1 cache is probably still an order of magnitude slower than CPU registers (I think more than 10 times slower). In addition, the `restrict` keyword minimizes not only the speed of the operations but their number as well, which could be meaningful... I mean, after all, the fastest operation is no operation at all :) – Myst Oct 24, 2019 at 20:40

14

Strict aliasing is not allowing different pointer types to the same data.

This article should help you understand the issue in full detail.

Share Follow

edited Nov 4, 2013 at 13:38
godel9
7,390 ● 1 ● 35 ● 54

answered Sep 19, 2008 at 1:33
Jason Dagit
13.8k ● 9 ● 35 ● 56

5 You can alias between references and between a reference and a pointer as well. See my tutorial dbp-consulting.com/tutorials/StrictAliasing.html – phorgan1 Aug 16, 2011 at 8:47

5 It is permitted to have different pointer types to the same data. Where strict aliasing comes in is when the same memory location is written through one pointer type and read through another. Also, some different types are permitted (e.g. `int` and a struct which contains an `int`). – M.M Jan 12, 2015 at 23:11

-5

Technically in C++, the strict aliasing rule is probably never applicable.

Note the definition of indirection (* operator):

> The unary * operator performs indirection: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and **the result is an lvalue referring to the object** or function **to which the expression points**.

Also from the definition of glvalue

> A glvalue is an expression whose evaluation determines the identity of an object, (...snip)

So in any well defined program trace, a glvalue refers to an object. **So the so called strict aliasing rule doesn't apply, ever.** This may not be what the designers wanted.

Share Follow

answered Jul 9, 2018 at 3:24

curiousguy

8,234 ● 2 ● 43 ● 60

4 The C Standard uses the term "object" to refer to a number of different concepts. Among them, a sequence of bytes that are exclusively allocated to some purpose, a not-necessarily-exclusive reference to a sequence of bytes to/from which a value of a particular type *could be* written or read, or such a reference that *actually* has been or will be accessed in some context. I don't think there's any sensible way to define the term "Object" that would be consistent with all the way the Standard uses it. – supercat Jul 9, 2018 at 20:01

1 @supercat Incorrect. Despite your imagination, it is actually fairly consistent. In ISO C it is defined as "region of data

storage in the execution environment, the contents of which can represent values". In ISO C++ there is a similar definition. Your comment is even more irrelevant than the answer because all you mentioned are ways of *representation* to refer objects' *content*, while the answer illustrates the C++ concept (glvalue) of a kind of expressions that tightly relates to the *identity* of objects. And all aliasing rules are basically relevant to the identity but not the content.
– FrankHB Mar 13, 2020 at 12:21

---

1 @FrankHB: If one declares `int foo;`, what is accessed by the lvalue expression `*(char*)&foo`? Is that an object of type `char`? Does that object come into existence at the same time as `foo`? Would writing to `foo` change the stored value of that aforementioned object of type `char`? If so, is there any rule that would allow the stored value of an object of type `char` to be accessed using an lvalue of type `int`? – supercat Mar 13, 2020 at 14:35

@FrankHB: In the absence of 6.5p7, one could simply say that every region of storage simultaneously contains all objects of every type that could fit in that region of storage, and that accessing that region of storage simultaneously accesses all of them. Interpreting in such fashion the use of the term "object" in 6.5p7, however, would forbid doing much of anything with non-character-type lvalues, which would clearly be an absurd result and totally defeat the purpose of the rule. Further, the concept of "object" used everywhere other than 6.5p6 has a static compile-time type, but...
– supercat Mar 13, 2020 at 14:38

---

1 sizeof(int) is 4, does the declaration `int i;` create four objects of each character type `in addition to one of type `int`? I see no way to apply a consistent definition of "object" which would allow for operations on both `*(char*)&i` and `i`. Finally, there's nothing in the Standard that allows even a `volatile`-

qualified pointer to access hardware registers that don't meet the definition of "object". – supercat Mar 15, 2020 at 17:03