The provenance memory model for C

A years-long effort led by Kayvan Memarian and Peter Sewell from Cambridge University, UK, Martin Uecker from Graz University of Technology, Austria, and myself (from ICube/Inria, France) has guided the C community to accept a common vision of so-called *pointer provenance*, defining how to trace the origins of pointer values through program execution. Our provenance-aware memory object model for C provides a precise mathematical specification, in place of the ambiguity of these aspects of the current C standard. It has also stimulated and informed discussion of provenance in the broader C, C++, Rust, and compiler communities.

This work has finally resulted in the publication of an international standard, Technical Specification ISO/IEC TS 6010 (edited by Henry Kleynhans, Bloomberg, UK). With the goal of making modern information systems safer and more secure, this official technical specification provides direction to all stakeholders in the industry such that they can converge their platforms and tools.

In this article, I will try to explain what this is all about, namely on how a provenance model for pointers interferes with alias analysis of modern compilers. For those that are not fluent with the terminology or the concept we have a short intro what pointer aliasing is all about, a review of existing tools to help the compiler and inherent difficulties and then the proposed model itself. At the end there is a brief takeaway that explains how to generally avoid complications and loss of optimization opportunities that could result from mis-guided aliasing analysis.

Pointer aliasing and program optimization

We say that two pointer values `p` and `q` during the execution of a program *alias* if they point to the same object in memory.[1] To see that the question if two pointers alias has an influence on the optimization of code, let's consider the following simple example of an iterative function.[2]

It implements an approximation algorithm for the reciprocal `r := 1/a` of the value `a := *a_p` which doesn't use division.

```
 // Reciprocal approximation
//
// This interface is horrific, don't use it.
// This just serves as an artificial example
// for this article.

constexpr double ε  = 0x1P-24;
constexpr double Π⁻ = 1.0 - ε;
constexpr double Π⁺ = 1.0 + ε;

void recip(double* a_p, double* ř_p) {
    for (;;) {
        register double Π = (*a_p)*(*ř_p);
        if ((Π⁻ &lt; Π) &amp;&amp; (Π &lt; Π⁺)) {
            break;
        } else {
            (*ř_p) *= (2.0 - Π);
        }
    }
}
```

The function receives a pointer to a second value `ř := *ř_p` with a rough approximation for `r`. It then iteratively approaches `r` within a chosen precision `ε`: the current values `a` and `ř` are multiplied into a value `Π` and if that value is sufficiently close to `1.0` the iteration stops. If it is not, `ř` is corrected and the loop

continues.

What is interesting for our context of aliasing is that this function has two pointer arguments that both point to a value of the same type `double`. One of these pointer targets $*\check{r}_p$ is loaded from memory, modified and stored at each iteration. In total, the non-optimized function as specified above in each iteration has

- 3 load and 1 store operations,
- 2 comparisons,
- 1 logical operation, and
- 3 arithmetic operations.

So loads and stores from memory make up 4 of about 10 operations in total.

But wait, can't this be done better? Yes, obviously, a much better version of this could look as follows.

```
void recip⁺(double* aₚ, double* řₚ) {
    register double a = *aₚ;
    register double ř = *řₚ;
    for (;;) {
        register double Π = a*ř;
        if ((Π⁻ < Π) && (Π < Π⁺)) {
            break;
        } else {
            ř *= (2.0 - Π);
        }
    }
    *řₚ = ř;
}
```

That is, we load `a` and `ř` once, at the beginning, and then only update $*\check{r}_p$ when we have finished our computation. The hope here is that these new local variables

use “hardware registers” that can be used directly by the processor, without going through loads and stores from and to the program memory.

So roughly the optimized function saves us 40% of the operations. This means that the optimized function is in general much faster and achieves its goal by wasting less energy.

Unfortunately no C compiler can do this optimization automatically:

> The functions `recip` and `recip⁺` and not equivalent.

We didn’t see this, yet, because we failed to discuss an important feature of the original program; the pointers a_p and $\check{r}_p$ may either point

- to different objects, or
- to the same object.

Indeed, our optimized function `recip⁺` only covers the first of these possibilities and not the second: the second case provides a completely different algorithm for which I wouldn’t know any use or properties. But since the compiler can’t know which of these cases we have in mind (maybe both?) it cannot do an optimization that excludes the second.[3]

In general, loads and stores from memory are expensive operations, so a lot of effort in modern compiler frameworks goes into so-called alias analysis, that is in an detailed analysis of which pointers may alias each other (or not). Such alias analysis then can be used to mechanically prove whether a specific optimization is feasible or not.

> Good alias analysis of pointer targets is crucial for > optimization in modern compilers.

In the case of `recip` we see that our specification is not precise enough to know whether or not the second case (where the pointers alias), can be excluded. We also see that misguided assumptions about pointer aliasing can result in implementing a completely different algorithm. Or, in other words, they may result in a severe bug.

> *Mislead aliasing assumptions result in bugs.*

In one of our papers, we provide a long list of examples where pointer aliasing can lead to different interpretations by users and compilers, see also this paper here.

Note also that a previous attempt by the C committee to introduce stricter aliasing rules had turned into years of flame wars between alienated parts of the user community, compiler builders and committee members. A communication disaster that was very harmful for the industry in general and to the trust of the community in the C committee in particular.

Helping the compiler

Modern compilers have several mechanisms that help to deduce more information about the intended use cases for pointers and from there to provide correct optimizations. These mechanisms are

- Type-based alias analysis.
- Flow-based alias analysis.

For type-based alias analysis, C takes a relatively simple approach, at least on the surface level.

> *Two pointers that point to objects of different non-character type > are supposed not to alias.*

So, for our function `recip` above, it would have been sufficient to have arguments pointing to differently typed objects. Then, an optimization similar to `recip+` would have been valid.

But ...

... notice the "supposed" in the phrase above. In C, it is entirely the programmers responsibility to ensure that two differently typed objects are in fact different. If you are casting spells to convert pointers from one target type to another, you have to prove for yourself that you may pass such pointers as arguments to functions without causing problems, there. You are completely on your own.

Flow-based analysis in C is supported by several tools

- The `register` storage class.
- The `restrict` pointer qualifier.
- The `volatile` qualifier on the target type.

The first ensures that the address of a so-declared object is never taken. And in C, when there are no pointers, there is no aliasing.[4] Although this is easy enough to use and provides feedback if one tries to obtain the address of such an object, it seems that this is much less used than it should.

The second is much more subtle. If a pointer parameter of a function is declared with `restrict`, every user of the function has to ensure that the pointer value is the unique view on the object from within for each call to the function. The definition in the standard of that feature is obscure and, again, it has the same drawback as type-based alias analysis in C: you are completely on your own.

The third is a feature that imposes that each access to a `volatile` qualified object reloads the object from memory, regardless what the compiler might know about its previous value. Using this feature excessively is like stepping on the break:

basically all optimization opportunities concerning that object are lost.

As extensions, some compilers also have command line flags that help the user to steer through. For example gcc has the options `-fstrict-aliasing` and `-fno-strict-aliasing` to switch aliasing analysis as defined by the C standard on or off.

And then, there is provenance …

In the current C standard provenance is only a hidden concept, the word "provenance" appears in it exactly 0 times. Nevertheless, it marks an important assumption for existing C, namely that somehow compilers are allowed to assume that two pointers don't alias when they are known to come from different "sources" in the program.

For programmers, the current state is a disaster. In many cases there is no way to know which assumptions a compiler makes:

- There is no way (but `restrict` and types, see above) to claim that two pointers never alias.
- It is difficult (but for `volatile`) to specify that they might alias such as to force optimization to be more restricted, either. This even in situations where, for example, the base types are in fact different, but where the user knows that the underlying object might still be the same.

The concept of provenance created many difficulties because it was left undefined exactly what it meant, so each compiler implementer had their own ideas how to assert that two pointers do (or don't) alias, and how to use that in optimization.

When digging deeper we observed several problem spots, and it took us in fact a long time to understand the different assumptions.

OBJECT GRANULARITY

The first question that comes up is to figure out at which granularity we can do or want to do aliasing analysis. At the level of bytes, words, basic type objects, structure objects, memory allocations, or, the whole address space?

The main problem here is that in C we can move from one pointer address to the next via arithmetic. The easiest examples are arrays. If we have access to an element in the middle of an array, we can move back and forth by simply adding or subtracting an integer value to the corresponding pointer value. How would we (the programmers) or the compiler know if two pointers point into the some array but at different elements? What happens if we start to step backwards from such pointers?

This picture becomes even more confusing, if we allow pointer arithmetic on the byte level. In C, the `offsetof` macro allows us to access arbitrary members of a structure by using such pointer arithmetic.

POINTER EQUALITY AND OBJECT LIFE TIME

When you are programming with pointers in C, you are hopefully aware of the "dangling pointer" problem. This refers to the fact that for example a pointer to a local variable becomes invalid once you leave the scope (e.g. function) where it is declared.

For aliasing analysis, this becomes even more complicated: not only does a pointer become invalid, the address of the dead object may be reused in a different context. So two pointers may even be equal, one pointing to a dead object, the other to a new one.

There is another situation where two pointer values are the same, but where they talk about two different objects. This occurs when two arrays, `A` and `B`, say,

happen to live in adjacent memory locations. If `A` has `n` elements, the pointer value `&A[0]+n` (the end of `A`) might be the same as `&B[0]` (the start of `B`). So then we have two pointer values that are the same but that have quite a different meaning for the programmer.

INFORMATION FLOW FOR ADDRESSES

Another concept influences aliasing analysis a lot, namely the possibility that a pointer value escapes from a limited scope of the program (e.g the address of a local variable) and becomes known in other parts of the program.

For example a variable that is declared `static` in a block is usually only visible there. In general the compiler masters very well which pointers may alias with a pointer value that corresponds to the address of that `static` variable. If the address of that variable "escapes" from the local scope, for example by passing it as an argument to a function, the address could come back into the same scope via unknown paths that are difficult to control.

There are surprisingly many ways that pointer values as a whole or parts of them may escape from one context and reappear in another. Among them are

- manipulations of the byte-representations of pointers for example
  - copying by `memcpy`,
  - copying by bytes,
  - IO using `%p` in `printf` or
  - `fwrite`
- manipulation of integers that are the result of pointer-to-integer conversions.

Perhaps surprisingly the latter is an important point that had to be handled carefully in our proposal. The reason is that pointer bit-manipulations are used in contexts where the available memory relatively constrained, such as systems

programming for example. Bit-manipulation tricks are then used to save on the size of data structures. This has not only the advantage that storage space for the data can be reduced, but also that a reduced size also improves the performance for data that are used intensively due to improved use of processor caches.

A famous example is the XOR trick for doubly linked lists, where a data structure stores the XOR of the bit patterns of two pointer values. So for that particular use we have an integer (here of type `uintptr_t`) that contains information that comes from two different pointers, that then is used to sometimes reconstruct one or the other of these pointer values.

```
 typedef struct elem elem;
struct elem {
    uintptr_t both;
    double data;
};

void elem_store(elem* e, elem* prev, elem* next, double val) {
    (*e) = (elem){
        both = (uintptr_t)prev ^ (uintptr_t)next,
        data = val,
    };
}
```

Such a data structure can then be used for elements of a list that can be traversed consistently forward and backward, but the memory footprint is only that of a simply-linked list:

```
 elem* elem_next(elem const* e, elem* prev) {
    return (elem*)((uintptr_t)prev ^ e-&gt;both);
}

elem* elem_prev(elem const* e, elem* next) {
    return (elem*)((uintptr_t)next ^ e-&gt;both);
```

```
}
```

As you can see, besides their names these two functions are completely identical, and I can't imagine any compiler being able to track the origin of a pointer that one of these two functions returns.

Tracking provenance through integers?

The fact that in C pointer values are closely related to integers creates a lot of confusion. In the case of aliasing analysis, the lack of separation between those terms has it that we have to integrate a model of information flow of pointer values (or just some bits or bytes of them) through integers and back to pointer values.

When Peter and Kayvan started their investigations, they had to consider different possibilities, one of them being to track information about pointers through such a chain of conversions. It turned out, that such a model would be possible (they provided a sound specification for it) but that it would come at an important cost. For code as for the XOR trick used above a pointer value (the result of a call to `elem_next` for example) would have two origins (`prev` and `next` in a call to `elem_store`) and not only one. Since such different origins could then accumulate if we do more operations, basically an integer and a pointer derived from it, could have an arbitrary number of origins.

Such a model with multiple origins of pointers seemed complicated and impractical. Complicated for users, because they would have to be aware that information about pointers could be used in surprising ways by a sophisticated compiler. Impractical for compilers because keeping track of all possible in-flow of information would result in a combinatorial explosion of the state of the abstract machine.

Presented with such a complication, one possibility would have been to just "forbid" using pointers in that way. We could have stated something along the lines of "*if a pointer has several origins, the behavior is undefined*" and thus leaving everything (the "undefined" part) to compiler implementations. But because these situation appear in real life code, this would have left these important parts non-portable between different compilers and architectures.

So the overall conclusion was not to ban such usage of pointers through integers, but to formalize it and label such pointers as *exposed*, that is that no user has to fear that compilers will present them with optimization that uses knowledge of such information flow, and no compiler has the pressure to optimize such code, either.

The provenance model

The provenance model that we came up with, and which is at the base of TS 6010, tries to take all of these aspects into account with the goal to provide something that at the same time can easily be referred to by users and compiler implementers. It provides some compromise between the expectations of the two communities in the sense that it does not leave all the liberty they might dream of to compiler providers for optimization, and it still has some sort of complexity and difficulty for users.

In the following we present the most important parts of that model.

Storage instance

As said above, we have to agree upon the granularity of memory accesses for which aliasing of pointers will be considered. When we combed through the existing standard (C11 an C17 at the time) we quickly noticed that there was not even agreement within that standard. When it talks about what is found at the

other end of a pointer, it talks about “object”,[5] “space”, “memory” or “storage” and even some combinations of these.

It seemed important to us to emphasize that pointers and addresses are already an abstraction that does not necessarily denote a physical device: most modern platforms nowadays form so-called virtual address spaces. Such “virtual addresses” then are in general transformed by low-level tools to “physical addresses” that represent concrete memory hardware. To make that distinction clearer we decided to use the term “storage” in most places where one of the terms noted above appear.

Another important observation to have is that we even have to talk about things that do not have an address. For example if we declare a variable width the `register` storage class, we cannot receive a pointer to that object and the whole point is that it is not necessary for the compiler to realize this variable in the main memory.

Then, the aspect of temporality also comes into play. A chunk of memory that is obtained for example by a call to `malloc` can be returned to the system by calling `free` and then might again be served by another call to `malloc`. It is important that the two entities to which the pointer refers are seen as completely different and that the fact that they reside in the same memory location is a simple coincidence.

For the granularity, we decided to go on the level of maximal region in which “legally” a C program could operate. Since inside any allocation or declared object all bytes are reachable by character-pointer arithmetic, we decided to take this as the level of granularity. Therefore

> A *storage instance* is the maximal region of storage that is > provided by > – an allocation (`malloc` and similar), > – a variable definition, > – a compound literal, or > – an object with temporary lifetime.

Note here that the second point talks about the definition of a variable, not its declaration. For local variables these two coincide, but for file scope variables (outside any function) there can be declarations (with `extern`) that are not definitions. The definition of a variable is always unique and specifies *where* it is located, namely in our terminology here, where the storage instance that represents it comes to be.

> If a storage instance is addressable, the conversion of a pointer to > its start to the type `unsigned char*` points into an array, called > its *byte array*, where each element is one byte of the storage > instance.

By that definition, conversions of pointers to character types and to `void*` are defined and it is uniquely prescribed how arithmetic on a byte level works.

> A storage instance has a lifetime that expands > – from the allocation (typically `malloc`) to the deallocation > (typically `free`) > – from the definition of the variable to the point where the block of the > definition left (for a VLA) > – from the point of entering the block of the definition until it is > left (for other variables with automatic storage duration) > – from the point of entering the innermost block that contains the > expression until that scope is left (for compound literals with > automatic storage duration) > – from the start of the program execution until its end (for `static` > objects) > – from the start of the thread execution until its end (for > `thread_local` objects) > – during the evaluation of the full expression that contains it (for > objects of temporary lifetime).

So in addition to the "where" above, this definition describes *when* the storage instance that represents an object comes to be and ceases to exist.

If you are not familiar with all the concepts in that item list, just ignore these. The importance here really is to make it clear for the features that you use in your program and know about, that in general their lifetime is limited and that any such allocation or definition gives rise to one single storage instance per context

in which the construct is met.

For example in the following code we see three storage instances in action

```
{
    size_t n  = 32;
    double (*A)[n][n] = malloc(sizeof(*A));
    ...
    free(A);
    A = nullptr;
    ...
}
```

The two obvious ones are

- The one for `n`, a variable of integer type, and for which the lifetime starts when we enter the block at the `{` and that ends when we leave it at `}`.
- The storage instance that is allocated by the call to `malloc` and that is deallocated by the call to `free`.

But then there is also a storage instance for the pointer `A` itself that, similar to `n`, lives during the execution of the block. In particular, after we freed the contents of `A` we may still access it to set it to a null pointer value.

For a case where the notion of storage instance is perhaps a bit less intuitive we note that calls to `realloc` are a bit peculiar with respect to that definition. In a call

```
void* p = realloc(q, 77);
```

we first have the storage instance to which `q` points. Then, if the call is

successful, that old storage instance is released and a pointer to a new storage instance is stored in `p`. Even if these two pointers are identical (possibly the storage instances start a the same address) they are nevertheless considered as two different entities.

With the term storage instance immediately comes the notion of provenance.

> The *provenance* of a valid pointer value is the storage instance > into which (or one beyond which) the pointer value points.

With the exception of one particular border case (see below) the provenance of a valid pointer value is unique.

Address space

To be useful in an aliasing model, the concept of an address space is not provided with enough precision in the current C standard. We need to talk consistently about addresses, how pointers convert, compare or relate.

The model we came up with, has the following properties:

- To each object pointer value corresponds an abstract address that is a positive integer value.
- Bytes within an addressable storage instance have abstract addresses that increase from start to end.
- If the distinct storage instances *A* and *B* are alive at a common point in time, the abstract addresses of all bytes of *A* are either all smaller or all greater than the abstract addresses of all bytes of *B*.
- Two pointer values are equal if they correspond to the same abstract address.
- One pointer value is smaller than another pointer value, if both point into the same storage instance and if the address of the first is smaller than the one of

the second.
- If the platform is able to represent all addresses in some integer type, the type `uintptr_t` is provided and a conversion from a pointer to that type provides the abstract address.
- Conversion from pointers to any integer type are consistent with that mapping to abstract addresses.

This model falls short from defining a "flat" address space:

- Arithmetic on pointers and arithmetic on abstract addresses need not to be consistent.
- Even within the same storage instance, the increase from one byte to next may not be one, and may not even be uniform.
- The type `uintptr_t` may not exist.
- Conversion to integer types that are too narrow has undefined behavior.

The reasons for only having such a lax definition are simple, for each of the weird properties in the list there are examples that make it necessary. In particular, there are platforms with segmented memories that have "bumps" in the address space, and platforms that pack additional bits into pointer values that are not related to the corresponding abstract address.

Exposure and synthesis of pointer values

Another observation is crucial for our model: most aliasing analysis isn't perfect. That is, compilers as well as programmers have limits of which tracks of the pointer information they can follow. For example the XOR trick shows that a pointer value can have several origins. In all we have to foresee a mechanism that describes the boundaries of the assumption that a compiler may make on one hand, and the guarantees that a programmer has to give on the other.

The mechanism with which we came up has two sides

- A pointer value is *exposed* if information about its abstract address or its in-memory representation leaks to the outside or to distant parts of the program.
- A pointer value is *synthesized* if it is assembled from outside information, from byte information or from integer values.

The goal is to describe that mechanism in a way such that (in principle) some auxiliary information could be added to each pointer value that would either allow

- compilers and users to establish aliasing properties of a set of pointer values, or
- easily come to a consensus for situations for which such analysis is abandoned an may not be assumed.

EXPOSURE

Let's now have a look at a possible normative text as it should be integrated into the C standard at some point

> A storage instance becomes *exposed* when a pointer value `p` of effective > type `T*` with this provenance is used in the following contexts: > – Any byte of the object representation of `p` is used in an > expression. > – The byte array pointed-to by the first argument of a call to the > `fwrite` library function intersects with an object representation of > `p`. > – `p` is converted to an integer. > – `p` is used as an argument to a `%p` conversion specifier of the > `printf` family of library functions.

The idea of the first bullet point is that if we read bytes of the object representation of a pointer, cascaded `if/else` control flow could be used to reconstruct pointers and thus jeopardize any aliasing analysis. But what we also didn't wanted, is that "normal" operations that programmers do on pointer

representations as a whole would have similar effect as access.

Therefore notice that `memcpy` or similar functions do not appear in the list above; as long as we use it to copy pointer representations as a whole, provenance can simply be transferred. For example, using `memcpy` on structure objects that have pointer members is fine: such an operation copies the whole pointer without exposing individual bytes. So we simply assume that the provenance information is transferred at the same time to the target structure.

We also don't want that small changes in the way that we look at a pointer representation has an influence on aliasing analysis. Therefore we add the following paragraph

> Nevertheless, if the object representation of `p` is read through an > lvalue of a pointer type `S*` that has the same representation and > alignment requirements as `T*`, that lvalue has the same provenance > as `p` and the provenance does not thereby become exposed.

Here the term "same representation and alignment" covers for example the possibility to look at

- a pointer that has different qualifiers than the original,
- where one type would be a signed type and the other unsigned, or
- one would be a structure and the other would be another structure that sits at the beginning of the first.

Also exposure is meant to be a one-way street, once exposed we will never know where the information about the pointer could creep into our program execution.

> Exposure of a storage instance is irreversible and constitutes a > side effect in the abstract state machine.

SYNTHESIS

The inverse operation that uses other information to produce a pointer looks as follows:

> A pointer value `p` is synthesized if it is constructed by one of the > following: > – Any byte of the object representation of `p` is changed > * by an explicit byte operation, > * by type punning with a non-pointer object or with a pointer object > that only partially overlaps, > * or by a call to `memcpy` or similar function that does not write > the entire pointer representation or where the source object does > not have an effective pointer type. > – The object representation of `p` intersects with a byte array > pointed-to by the first argument of a call to the `fread` library > function. > – `p` is converted from an integer value. > – `p` is used as an argument to a `%p` conversion specifier of the > `scanf` family of library functions.

Additionally, we always require that the storage instance that is synthesized has been exposed before.

Ambiguities

With all of that the situation about adjacent storage instances still remains. That is, a situation where two arrays `A` and `B` are adjacent in memory. Let's suppose the two arrays have two elements, each, and that the base type has four bytes:

A[0]	A[1]	B[0]	B[1]
a_0 a_1 a_2 a_3	a_4 a_5 a_6 a_7	b_0 b_1 b_2 b_3	b_4 b_5 b_6 b_7

When we are taking addresses the following are valid expressions: `&A[2]`, which points to the element just after the array `A`, and `&B[0]`, which points to the beginning of `B`. So if `A` and `B` are adjacent objects, the pointer value of the

first expression is exactly the same as for the second. So both represent different semantics but with an abstract address that is the same, the byte address of byte b_0.

So far, so good. Using these pointers with clearly indicated semantics doesn't pose a problem for aliasing analysis. In particular in our model a pointer value as given above always has a unique provenance. A problem arises only if

- The storage instance of `A` and `B` have both been exposed, by a conversion to `uintptr_t`, say.
- A pointer `Λ` is synthesized that corresponds to the byte address of b_0 by converting some `uintptr_t` value back.

Then we are in the dilemma that `Λ` could have both provenances, that of `A` or that of `B`.

Already from the length of the text that is needed to describe the situation you might guess that this is a very rare situation, the easiest remedy against its difficulties is just not to have it in the first place. But in the case that it arises we have to foresee a mechanism that is consistent with the model. Since there is generally no additional information available that could guide the compiler to see which semantics the programmer meant, the semantics are deduced from the usage of the pointer value:

> A synthesized pointer value `Λ` with two possible provenances `A` or > `B` is assumed to have the one provenance among the two that is > consistent with its use in expressions.

That is for example, if we use `Λ` in

- `Λ[0]`, `*Λ` or `Λ+1` it is assumed that the provenance is the one of `B`
- `Λ[-1]` or `Λ-1` it is assumed that the provenance is the one of `A`.

A use as in `Λ+0` that has the same value `Λ` gives no indication of a direction in which we want to step through the array. So it does not fix a choice of one provenance or the other. But `Λ+1-1` which resolves to `(Λ+1)-1` has the provenance of `B` and similarly `Λ-1+1 ≡ (Λ-1)+1` has the one of `A`.

So if and when you have to use that marginal constructs, make sure that you give clear indications to the compiler into which of the two storage instance you want to walk.

Takeaway

If you were brave enough to follow this article up to here, you probably deserve some reassuring words such that you are not left alone with a complicated web of choices and interrelationships between parts of your code that are impossible to master. In the contrary, our model provides a very simple method to guarantee sound aliasing analysis by compilers of your every day code:

> *Do not expose pointer values ...*

... if you can avoid it. And in most cases you can. In particular avoid

> – taking the address of variables (maybe use `register` to be sure), > – casts of pointer values to and from integers, > – accessing individual bytes of pointer targets (AKA coversion to > character pointers), > – conversions of pointer values to any other unrelated pointer type, > – accessing individual bytes of byte-representations of pointer values, > – printing pointer values with `%p` or by using `fwrite`, > – using the end address of an array to walk backwards into the array,

and you will be fine.

Obviously, these features are important for C and contribute for a lot of its power.

But you should only use them when (and where) you master them in all of their consequences. For example, if in the context of system's programming you need the XOR trick for your doubly-linked list, that is fine as long as you are aware, that this might cost you some other optimization opportunities. Or if you are debugging your code, printing pointer values with `%p` can be crucial, but you should make sure that you disable all traces of such printing when you compile for production.

Generally, using more modern features of C will help your compiler to provide you with more efficient and safer executables. For example, in addition to the above

- not using pointer parameters on functions when you are only interested in the value (our `recip` function is a bad example),
- using `const` qualification wherever you may,
- not using casts,
- not using integer zero as a null pointer,
- using signed and unsigned integer types consistently,

all contribute to help your C compiler to do at what it's best: nagging you about things that you might have overlooked.

1. Or more precisely if the objects pointed-to have at least one byte in common. ↩

2. Please, please, don't take the function as given here as a basis for your code. It is only chosen for illustration of the points that I want to make here about aliasing. ↩

3. Any modern compiler will still do some optimization with `recip`, but would usually not arrive at the same level of improvement. ↩

4. This might be completely different in other programming languages. E.g in C++ there is a concept of “references”, that when passed around into functions, result in similar questions on aliasing as we see here for pointers in C. ↩

5. Confusingly, there is another notion of object related to types that is used throughout the C standard, but which is not the same as an “allocated object” as we are talking, here. ↩

Jens Gustedt / June 30, 2025 / C17, C23, compiler optimization, Modern C