C User's Guide Supplement

Forte Developer™ 6 update 1
(Sun WorkShop™ 6 update 1)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 806-6145-10
October 2000, Revision A

Send comments about this document to: docfeedback@sun.com

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Type-Based Alias Analysis

This document explains how you can use a new Sun WorkShop Compilers C option and several new pragmas to enable the compiler to perform type-based alias analysis and optimizations. You use these extensions to express type-based information about the way pointers are used in your C program. The C compiler uses this information, in turn, to do a significantly better job of alias disambiguation for pointer-based memory references in your program.

This document contains the following sections:

Introduction

You can use the new `-xalias_level` option to specify one of seven alias levels. Each level specifies a certain set of properties about the way you use pointers in your C program.

As you compile with higher levels of the `-xalias_level` option, the compiler makes increasingly extensive assumptions about the pointers in your code. You have greater programming freedom when the compiler makes fewer assumptions. However, the optimizations that result from these narrow assumptions may not result in significant runtime performance improvement. If you code in accordance with the compiler assumptions of the more advanced levels of the `-xalias_level` option, there is a greater chance that the resulting optimizations will enhance runtime performance.

The `-xalias_level` option specifies which alias level applies to each translation unit. For cases where more detail is beneficial, you can use new pragmas to override whatever alias levels are in effect so that you can explicitly specify the aliasing relationships between individual types or pointer variables in the translation unit. These pragmas are most useful when the pointer usage in a translation unit is covered by one of the available alias levels, but a few specific pointer variables are used in an irregular way that is not allowed by one of the available levels.

The `-xalias_level` Option

The compiler uses the `-xalias_level` option to determine what assumptions it can make in order to perform optimizations using type-based alias analysis. This option places the indicated alias level into effect for the translation units being compiled. The first default is `-xalias_level=any`, which means there is no type-based alias analysis. If you specify `-xalias_level` without a value, the default is `-xalias_level=layout`.

Remember that if you issue the `-xalias_level` option but you fail to adhere to all of the assumptions and restrictions about aliasing described for any of the alias levels, the behavior of your program is undefined.

-xalias_level=*list*

Replace *list* with one of the terms in the following table.

Term	Meaning
any	The compiler assumes that all memory references can alias at this level. There is no type-based alias analysis at the level of -xalias_level=any.
basic	If you use the -xalias_level=basic option, the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler also assumes that references to all other types can alias each other as well as any C basic type. The compiler assumes that references using char * can alias any other type. For example, at the -xalias_level=basic level, the compiler assumes that a pointer variable of type int * is not going to access a float object. Therefore it is safe for the compiler to perform optimizations that assume a pointer of type float * will not alias the same memory that is referenced with a pointer of type int *.
weak	If you use the -xalias_level=weak option, the compiler assumes that any structure pointer can point to any structure type. Any structure or union type that contains a reference to any type that is either referenced in an expression in the source being compiled or is referenced from outside the source being compiled, must be declared prior to the expression in the source being compiled. You can satisfy this restriction by including all the header files of a program that contain types that reference any of the types of the objects referenced in any expression of the source being compiled. At the level of -xalias_level=weak, the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using char * alias memory references that involve any other type.
layout	If you use the -xalias_level=layout option, the compiler assumes that memory references that involve types with the same sequence of types in memory can alias each other. The compiler assumes that two references with types that do not look the same in memory do not alias each other. The compiler assumes that any two memory accesses through different struct types alias if the initial members of the structures look the same in memory. However, at this level, you should not use a pointer to a struct to access some field of a dissimilar struct object that is beyond any of the common initial sequence of members that look the same in memory between the two structs. This is because the compiler assumes that such references do not alias each other. At the level of -xalias_level=layout the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using char * can alias memory references involving any other type.

Term	Meaning
`strict`	If you use the `-xalias_level=strict` option, the compiler assumes that memory references, that involve types such as structs or unions, that are the same when tags are removed, can alias each other. Conversely, the compiler assumes that memory references involving types that are not the same even after tags are removed do not alias each other. However, any structure or union type that contains a reference to any type that is part of any object referenced in an expression in the source being compiled, or is referenced from outside the source being compiled, must be declared prior to the expression in the source being compiled. You can satisfy this restriction by including all the header files of a program that contain types that reference any of the types of the objects referenced in any expression of the source being compiled. At the level of `-xalias_level=strict` the compiler assumes that memory references that involve different C basic types do not alias each other. The compiler assumes that references using `char *` can alias any other type.
`std`	If you use the `-xalias_level=std` option, the compiler assumes that types and tags need to be the same to alias, however, references using `char *` can alias any other type. This rule is the same as the restrictions on the dereferencing of pointers that are found in the 1999 ISO C standard. Programs that properly use this rule will be very portable and should see good performance gains under optimization.
`strong`	If you use the `-xalias_level=strong` option, the same restrictions apply as at the `std` level, but additionally, the compiler assumes that pointers of type `char *` are used only to access an object of type char. Also, the compiler assumes that there are no interior pointers. An interior pointer is defined as a pointer that points to a member of a struct.

Pragmas

For cases in which type-based analysis can benefit from more detail, you can use the following pragmas to override the alias level in effect and specify the aliasing relationships between individual types or pointer variables in the translation unit. These pragmas provide the most benefit when the use of pointers in a translation unit is consistent with one of the available alias levels, but a few specific pointer variables are used in an irregular way not allowed by one of the available levels.

Note – You must declare the named type or variable prior to the pragma or a warning message is issued and the pragma is ignored. The results of the program are undefined if the pragma appears after the first memory reference to which its meaning applies.

The following terms are used in the pragma definitions.

Term	Meaning
level	Any of the alias levels listed under "The `-xalias_level` Option" on page 2.
type	Any of the following: • `char`, `short`, `int`, `long`, `long long`, `float`, `double`, `long double` • `void`, which denotes all pointer types • `typedef` *name*, which is the name of a defined type from a `typedef` declaration • `struct` *name*, which is the keyword `struct` followed by a *struct tag* name • `union`, which is the keyword `union` followed by a *union tag* name
pointer_name	The name of any variable of pointer type in the translation unit.

#pragma alias_level *level list*

Replace *level* with one of the seven alias levels: `any`, `basic`, `weak`, `layout`, `strict`, `std`, or `strong`. You can replace *list* with either a single type or a comma-delimited list of types, or you can replace *list* with either a single pointer or a comma-delimited list of pointers. For example, you can issue `#pragma alias_level` as follows:

- `#pragma alias_level` *level* `(type [, type])`
- `#pragma alias_level` *level* `(pointer [, pointer])`

This pragma specifies that the indicated alias level applies either to all of the memory references of the translation unit for the listed types, or to all of the dereferences of the translation unit where any of the named pointer variables are being dereferenced.

If you specify more than one alias level to be applied to a particular dereference, the level that is applied by the pointer name, if any, has precedence over all other levels. The level applied by the type name, if any, has precedence over the level applied by the option. In the following example, the `std` level applies to *p* if the program is compiled with `#pragma alias_level` set higher than `any`.

```
typedef int * int_ptr;
int_ptr p;
#pragma alias_level strong (int_ptr)
#pragma alias_level std (p)
```

#pragma alias (type, type [, type]...)

This pragma specifies that all the listed types alias each other. In the following example, the compiler assumes that the indirect access `*pt` aliases the indirect access `*pf`.

```
#pragma alias (int, float)
int *pt;
float *pf;
```

#pragma alias (pointer, pointer [, pointer]...)

This pragma specifies that at the point of any dereference of any of the named pointer variables, the pointer value being dereferenced can point to the same object as any of the other named pointer variables. This pragma overrides the aliasing assumptions of any applied alias levels. In the following example, the compiler assumes that the indirect access `*p` aliases the indirect access `*q` regardless of the types of the two pointers.

```
#pragma alias(p, q)
```

#pragma may_point_to (pointer, variable [, variable]...)

This pragma specifies that at the point of any dereference of the named pointer variable, the pointer value being dereferenced can point to the objects that are contained in any of the named variables. This pragma overrides the aliasing assumptions of any applied alias levels. In the following example, the compiler assumes that the indirect access `*p` aliases the direct accesses `a`, `b`, and `c`.

```
#pragma alias may_point_to(p, a, b, c)
```

#pragma noalias (type, type [, type]...)

This pragma specifies that the listed types do not alias each other. In the following example, the compiler assumes that the indirect access `*p` does not alias the indirect access `*ps`.

```
struct S {
   float f;
   ...} *ps;

#pragma noalias(int, struct S)
int *p;
```

#pragma noalias (pointer, pointer [, pointer]...)

This pragma specifies that at the point of any dereference of any of the named pointer variables, the pointer value being dereferenced does not point to the same object as any of the other named pointer variables. This pragma overrides all other applied alias levels. In the following example, the compiler assumes that the indirect access `*p` does not alias the indirect access `*q` regardless of the types of the two pointers.

```
#pragma noalias(p, q)
```

#pragma may_not_point_to (pointer, variable [, variable]...)

This pragma specifies that at the point of any dereference of the named pointer variable, the pointer value being dereferenced does not point to the objects that are contained in any of the named variables. This pragma overrides all other applied alias levels. In the following example, the compiler assumes that the indirect access `*p` does not alias the direct accesses `a`, `b`, or `c`.

```
 #pragma may_not_point_to(p, a, b, c)
```

Examples

This section provides examples of how memory references are constrained in your program when you use the `-xalias_level` option.

Example 1

Consider the following code example that can compile with different levels of aliasing to demonstrate the aliasing relationship of the shown types.

```
struct foo {
    int f1;
    short f2;
    short f3;
    int f4;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;

int *ip;
short *sp;
```

If Example 1 is compiled with the `-xalias_level=any` option, the compiler considers the following indirect accesses as aliases to each other:

`*ip`, `*sp`, `*fp`, `*bp`, `fp->f1`, `fp->f2`, `fp->f3`, `fp->f4`, `bp->b1`, `bp->b2`, `bp->b3`

If Example 1 is compiled with the `-xalias_level=basic` option, the compiler considers the following indirect accesses as aliases to each other:

`*ip`, `*bp`, `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, `bp->b3`

Additionally, `*sp`, `fp->f2`, and `fp->f3` can alias each other, and `*sp` and `*fp` can alias each other.

However, under `-xalias_level=basic`, the compiler assumes the following:

- `*ip` does not alias `*sp`.
- `*ip` does not alias `fp->f2` and `fp->f3`.
- `*sp` does not alias `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3`.

The compiler makes these assumptions because the access types of the two indirect accesses are different basic types.

If Example 1 is compiled with the `-xalias_level=weak` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2` and `fp->f3`.
- `fp->f1` can alias `bp->b1.`
- `fp->f4` can alias `bp->b3.`

The compiler assumes that `fp->fp1` does not alias `bp->b2` because `f1` is a field with offset 0 in a structure, whereas `b2` is a field with a 4-byte offset in a structure. Similarly, the compiler assumes that `fp->f1` does not alias `bp->b3`, and `fp->f4` does not alias either `bp->b1` or `bp->b2`.

If Example 1 is compiled with the `-xalias_level=layout` option, the compiler assumes the following information:

- `*ip` can alias `*fp`, `*bp`, `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.
- `fp->f1` can alias `bp->b1` and `*bp`.
- `*fp` and `*bp` can alias each other.

`fp->f4` does not alias `bp->b3` because `f4` and `b3` are not corresponding fields in the common initial sequence of `foo` and `bar`.

If Example 1 is compiled with the `-xalias_level=strict` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.

With `-xalias_level=strict`, the compiler assumes that `*fp`, `*bp`, `fp->f1`, `fp->f2`, `fp->f3`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3` do not alias each other because `foo` and `bar` are not the same when field names are ignored. However, `fp` aliases `fp->f1` and `bp` aliases `bp->b1`.

If Example 1 is compiled with the `-xalias_level=std` option, the compiler assumes the following alias information:

- `*ip` can alias `*fp`, `fp->f1`, `fp->f4`, `*bp`, `bp->b1`, `bp->b2`, and `bp->b3`.
- `*sp` can alias `*fp`, `fp->f2`, and `fp->f3`.

However, `fp->f1` does not alias `bp->b1`, `bp->b2`, or `bp->b3` because `foo` and `bar` are not the same when field names are considered.

If Example 1 is compiled with the `-xalias_level=strong` option, the compiler assumes the following alias information:

- `*ip` does not alias `fp->f1`, `fp->f4`, `bp->b1`, `bp->b2`, and `bp->b3` because a pointer, such as `*ip`, should not point to the interior of a structure.
- Similarly, `*sp` does not alias `fp->f1` or `fp->f3`.
- `*ip` does not alias `*fp`, `*bp`, and `*sp` due to differing types.
- `*sp` does not alias `*fp`, `*bp`, and `*ip` due to differing types.

Example 2

Consider the following example source code that can compile with different levels of aliasing to demonstrate the aliasing relationship of the shown types.

```
struct foo {
    int f1;
    int f2;
    int f3;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;
```

If Example 2 is compiled with the `-xalias_level=any` option, the compiler assumes the following alias information:

`*fp`, `*bp`, `fp->f1`, `fp->f2`, `fp->f3`, `bp->b1`, `bp->b2` and `bp->b3` all can alias each other because any two memory accesses alias each other at the level of `-xalias_level=any`.

If Example 2 is compiled with the `-xalias_level=basic` option, the compiler assumes the following alias information:

`*fp`, `*bp`, `fp->f1`, `fp->f2`, `fp->f3`, `bp->b1`, `bp->b2` and `bp->b3` all can alias each other. Any two field accesses using pointers `*fp` and `*bp` can alias each other in this example because all the structure fields are the same basic type.

If Example 2 is compiled with the `-xalias_level=weak` option, the compiler assumes the following alias information:

- `*fp` and `*fp` can alias each other.
- `fp->f1` can alias `bp->b1`, `*bp` and `*fp`.
- `fp->f2` can alias `bp->b2`, `*bp` and `*fp`.
- `fp->f3` can alias `bp->b3`, `*bp` and `*fp`.

However, `-xalias_level=weak` imposes the following restrictions:

- `fp->f1` does not alias `bp->b2` or `bp->b3` because `f1` has an offset of zero, which is different from that of `b2` (four bytes) and `b3` (eight bytes).
- `fp->f2` does not alias `bp->b1` or `bp->b3` because `f2` has an offset of four bytes, which is different from `b1` (zero bytes) and `b3` (eight bytes).
- `fp->f3` does not alias `bp->b1` or `bp->b2` because `f3` has an offset of eight bytes, which is different from `b1` (zero bytes) and `b2` (four bytes).

If Example 2 is compiled with the `-xalias_level=layout` options, the compiler assumes the following alias information:

- `*fp` and `*bp` can alias each other.
- `fp->f1` can alias `bp->b1`, `*bp`, and `*fp`.
- `fp->f2` can alias `bp->b2`, `*bp`, and `*fp`.
- `fp->f3` can alias `bp->b3`, `*bp`, and `*fp`.

However, `-xalias_level=layout` imposes the following restrictions:

- `fp->f1` does not alias `bp->b2` or `bp->b3` because field `f1` corresponds to field `b1` in the common initial sequence of `foo` and `bar`.
- `fp->f2` does not alias `bp->b1` or `bp->b3` because `f2` corresponds to field `b2` in the common initial sequence of `foo` and `bar`.
- `fp->f3` does not alias `bp->b1` or `bp->b2` because `f3` corresponds to field `b3` in the common initial sequence of `foo` and `bar`.

If Example 2 is compiled with the `-xalias_level=strict` option, the compiler assumes the following alias information:

- `*fp` and `*bp` can alias each other.
- `fp->f1` can alias `bp->b1`, `*bp`, and `*fp`.
- `fp->f2` can alias `bp->b2`, `*bp`, and `*fp`.
- `fp->f3` can alias `bp->b3`, `*bp`, and `*fp`.

However, `-xalias_level=strict` imposes the following restrictions:

- `fp->f1` does not alias `bp->b2` or `bp->b3` because field `f1` corresponds to field `b1` in the common initial sequence of `foo` and `bar`.
- `fp->f2` does not alias `bp->b1` or `bp->b3` because `f2` corresponds to field `b2` in the common initial sequence of `foo` and `bar`.
- `fp->f3` does not alias `bp->b1` or `bp->b2` because `f3` corresponds to field `b3` in the common initial sequence of `foo` and `bar`.

If Example 2 is compiled with the `-xalias_level=std` option, the compiler assumes the following alias information:

`fp->f1`, `fp->f2`, `fp->f3`, `bp->b1`, `bp->b2`, and `bp->b3` do not alias each other.

If Example 2 is compiled with the `-xalias_level=strong` option, the compiler assumes the following alias information:

`fp->f1`, `fp->f2`, `fp->f3`, `bp->b1`, `bp->b2`, and `bp->b3` do not alias each other.

Example 3

Consider the following example source code that demonstrates that certain levels of aliasing cannot handle interior pointers. For a definition of interior pointers see "`strong`" on page 4.

```
struct foo {
        int f1;
        struct bar *f2;
        struct bar *f3;
        int f4;
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

The dereference in Example 3 is not supported by `weak`, `layout`, `strict`, or `std`. After the pointer assignment `bp=(struct bar*)(&fp->f2)`, the following pair of memory accesses touches the same memory locations:

- `fp->f2` and `bp->b2` access the same memory location
- `fp->f3` and `bp->b3` access the same memory location
- `fp->f4` and `bp->b4` access the same memory location

However, under options `weak`, `layout`, `strict`, and `std`, the compiler assumes that `fp->f2` and `bp->b2` do not alias. The compiler makes this assumption because `b2` has an offset of zero, which is different from the offset of `f2` (four bytes), and `foo` and `bar` do not have a common initial sequence. Similarly, the compiler also assumes that `bp->b3` does not alias `fp->f3`, and `bp->b4` does not alias `fp->f4`.

Thus, the pointer assignment `bp=(struct bar*)(&fp->f2)`creates a situation in which the compiler's assumptions about alias information are incorrect. This may lead to incorrect optimization.

Try compiling after you make the modifications shown in the following example.

```
struct foo {
        int f1;
        struct bar fb;   /* Modified line */
#define f2 fb.b2         /* Modified line */
#define f3 fb.b3         /* Modified line */
#define f4 fb.b4         /* Modified line */
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

After the pointer assignment `bp=(struct bar*)(&fp->f2)`, the following pair of memory accesses touches the same memory locations:

- `fp->f2` and `bp->b2`
- `fp->f3` and `bp->b3`
- `fp->f4` and `bp->b4`

By examining the changes shown in the preceding code example, you can see that the expression `fp->f2` is another form of the expression `fp->fb.b2`. Because `fp->fb` is of type `bar`, `fp->f2` accesses the `b2` field of `bar`. Furthermore, `bp->b2` also accesses the `b2` field of `bar`. Therefore, the compiler assumes that `fp->f2` aliases `bp->b2`. Similarly, the compiler assumes that `fp->f3` aliases `bp->b3`, and `fp->f4` aliases `bp->b4`. As a result, the aliasing assumed by the compiler matches the actual aliases caused by the pointer assignment.

Example 4

Consider the following example source code.

```
struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

struct cat {
        int c1;
        struct foo cf;
        int c2;
        int c3;
} *cp;

struct dog {
        int d1;
        int d2;
        struct bar db;
        int d3;
} *dp;
```

If Example 4 is compiled with the `-xalias_level=weak` option, the compiler assumes the following alias information:

- `fp->f1` can alias `bp->b1`, `cp->c1`, `dp->d1`, `cp->cf.f1`, and `df->db.b1`.
- `fp->f2` can alias `bp->b2`, `cp->cf.f1`, `dp->d2`, `cp->cf.f2`, `df->db.b2`, `cp->c2`.
- `bp->b1` can alias `fp->f1`, `cp->c1`, `dp->d1`, `cp->cf.f1`, and `df->db.b1`.
- `bp->b2` can alias `fp->f2`, `cp->cf.f1`, `dp->d2`, `cp->cf.f1`, and `df->db.b2`.

`fp->f2` can alias `cp->c2` because `*dp` can alias `*cp` and `*fp` can alias `dp->db`.

- `cp->c1` can alias `fp->f1`, `bp->b1`, `dp->d1`, and `dp->db.b1`.
- `cp->cf.f1` can alias `fp->f1`, `fp->f2`, `bp->b1`, `bp->b2`, `dp->d2`, and `dp->d1`.

`cp->cf.f1` does not alias `dp->db.b1`.

- `cp->cf.f2` can alias `fp->f2`, `bp->b2`, `dp->db.b1`, and `dp->d2`.
- `cp->c2` can alias `dp->db.b2`.

`cp->c2` does not alias `dp->db.b1` and `cp->c2` does not alias `dp->d3`.

With respect to offsets, `cp->c2` can alias `db->db.b1` only if `*dp` aliases `cp->cf`. However, if `*dp` aliases `cp->cf`, then `dp->db.b1` must alias beyond the end of `foo cf`, which is prohibited by object restrictions. Therefore, the compiler assumes that `cp->c2` cannot alias `db->db.b1`.

`cp->c3` can alias `dp->d3`.

Notice that `cp->c3` does not alias `dp->db.b2`. These memory references do not alias because the offsets of the fields of the types involved in the dereferences differ and do not overlap. Based on this, the compiler assumes they cannot alias.

- `dp->d1` can alias `fp->f1`, `bp->b1`, and `cp->c1`.
- `dp->d2` can alias `fp->f2`, `bp->b2`, and `cp->cf.f1`.
- `dp->db.b1` can alias `fp->f1`, `bp->b1`, and `cp->c1`.
- `dp->db.b2` can alias `fp->f2`, `bp->b2`, `cp->c2`, and `cp->cf.f1`.
- `dp->d3` can alias `cp->c3`.

Notice that `dp->d3` does not alias `cp->cf.f2`. These memory references do not alias because the offsets of the fields of the types involved in the dereferences differ and do not overlap. Based on this, the compiler assumes they cannot alias.

If Example 4 is compiled with the `-xalias_level=layout` option, the compiler assumes only the following alias information:

- `fp->f1`, `bp->b1`, `cp->c1` and `dp->d1` all can alias each other.
- `fp->f2`, `bp->b2` and `dp->d2` all can alias each other.
- `fp->f1` can alias `cp->cf.f1` and `dp->db.b1`.
- `bp->b1` can alias `cp->cf.f1` and `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2` and `dp->db.b2`.
- `bp->b2` can alias `cp->cf.f2` and `dp->db.b2`.

If Example 4 is compiled with the `-xalias_level=strict` option, the compiler assumes only the following alias information:

- `fp->f1` and `bp->b1` can alias each other.
- `fp->f2` and `bp->b2` can alias each other.
- `fp->f1` can alias `cp->cf.f1` and `dp->db.b1`.
- `bp->b1` can alias `cp->cf.f1` and `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2` and `dp->db.b2`.
- `bp->b2` can alias `cp->cf.f2` and `dp->db.b2`.

If Example 4 is compiled with the `-xalias_level=std` option, the compiler assumes only the following alias information:

- `fp->f1` can alias `cp->cf.f1`.
- `bp->b1` can alias `dp->db.b1`.
- `fp->f2` can alias `cp->cf.f2`.
- `bp->b2` can alias `dp->db.b2`.

Example 5

Consider the following example source code.

```
struct foo {
        short f1;
        short f2;
        int   f3;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

union moo {
        struct foo u_f;
        struct bar u_b;
} u;
```

Here are the compiler's assumptions based on the following alias levels:

- If Example 5 is compiled with the `-xalias_level=weak` option, `fp->f3` and `bp->b2` can alias each other.
- If Example 5 is compiled with the `-xalias_level=layout` option, no fields can alias each other.
- If Example 5 is compiled with the `-xalias_level=strict` option, `fp->f3` and `bp->b2` can alias each other.
- If Example 5 is compiled with the `-xalias_level=std` option, no fields can alias each other.

Example 6

Consider the following example source code.

```
struct bar;

struct foo {
        struct foo *ffp;
        struct bar *fbp;
} *fp;

struct bar {
        struct bar *bbp;
        long        b2;
} *bp;
```

Here are the compiler's assumptions based on the following alias levels:

- If Example 6 is compiled with the `-xalias_level=weak` option, only `fp->ffp` and `bp->bbp` can alias each other.
- If Example 6 is compiled with the `-xalias_level=layout` option, only `fp->ffp` and `bp->bbp` can alias each other.
- If Example 6 is compiled with the `-xalias_level=strict` option, no fields can alias because the two struct types are still different even after their tags are removed.
- If Example 6 is compiled with the `-xalias_level=std` option, no fields can alias because the two types and the tags are not the same.

Example 7

Consider the following example source code:

```
struct foo;
struct bar;
#pragma alias (struct foo, struct bar)

struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        short b1;
        short b2;
        int   b3;
} *bp;
```

The pragma in this example tells the compiler that `foo` and `bar` are allowed to alias each other. The compiler makes the following assumptions about alias information:

- `fp->f1` can alias with `bp->b1`, `bp->b2`, and `bp->b3`
- `fp->f2` can alias with `bp->b1`, `bp->b2`, and `bp->b3`