C ユーザーズガイドの追補

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

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

Part No. 806-6490-01
2000 年 11 月 Revision A

原典： *C User's Guide Supplement*
Part No: 805-6145-10
Revision A

製品名の変更について

Sun は新しい開発製品戦略の一環として、Sun の開発ツール群の製品名を Sun WorkShop™ から Forte™ Developer に変更いたしました。製品自体の内容に変更はなく、従来通りの高品質をお届けいたします。

これまでの Sun の主力製品である基本プログラミングツールに、Forte Fusion™ や Forte™ for Java™ といった Forte 開発ツールの得意とする、マルチプラットフォームおよびビジネスアプリケーション実装の機能を盛り込むことで、より広範囲できめ細かな製品ラインが完成しました。

WorkShop 5.0 で使用されていた名称と、Forte Developer 6 で使用される新しい名称の対応については、以下の表をご覧ください。

旧名称	新名称
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

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

- Forte for High Performance Computing には Sun Performance WorkShop Fortran に含まれていたすべてのツール、および C++ コンパイラが含まれます。したがって、High Performance Computing のユーザーは開発用に 1 つの製品だけを購入すれば済むことになります。
- Forte Fortran Desktop Edition は以前の Sun Performance WorkShop Personal Edition と同じです。ただし、この製品に含まれる Fortran コンパイラでは、自動並列化されたコード、および明示的な指令に基づいた並列コードは生成できません。この機能は Forte for High Performance Computing に含まれる Fortran コンパイラでは使用できます。

Sun の開発製品を引き続きご利用いただきましてありがとうございます。今後もみなさまのご要望にお応えする製品をお届けできるよう努力してまいります。

目次

型に基づいた別名解析

このマニュアルでは、型に基づいた別名解析を使って最適化を行うための Sun WorkShop™ Compilers C の新しいオプションおよびいくつかのプラグマについて説明しています。この機能を使用すると、ポインタの使用方法に関する、型に基づいた情報を C プログラム中で表現することができます。C コンパイラにこの情報を使用させることで、プログラム中のポインタに基づいたメモリー参照の別名解析を、より効果的かつ効率的に行うことができます。

このマニュアルは次の節で構成されています。

はじめに

新しいオプション `-xalias_level` には、7 つある別名解析のレベルのうちの 1 つを指定できます。7 つのレベルごとに、C プログラム中でのポインタの使用方法に関する属性が定められています。

`-xalias_level` オプションで指定するレベルが高いほど、コンパイラはコード中でのポインタの使用方法に関する「仮定」の範囲を広げます。コンパイラの仮定が少ないほど、プログラミングの自由度は上がります。しかし、仮定が少ないと、最適化による実行時パフォーマンスの改善はあまり期待できません。`-xalias_level` オプ

ションを使ってコンパイラに広範囲に渡る仮定を行わせ、その仮定を踏まえてコーディングすることで、最適化によって実行時パフォーマンスを飛躍的に向上させることができます。

-xalias_level オプションは、各翻訳単位で適用される別名解析のレベルを指定します。より細かい単位でレベルを設定したい場合は、新しいプラグマを使用してください。このプラグマを使用すると、翻訳単位で現在有効になっている別名解析のレベルが何であれ、特定の型またはポインタ変数の関係の別名解析レベルを個別に指定できます。このプラグマは、翻訳単位全体には -xalias_level オプションのレベルが適用できるが、特定のポインタ変数が通常とは異なる方法で使用されているため、そのポインタ変数だけがオプションのレベルに準拠していない、といった場合に便利です。

-xalias_level オプション

コンパイラは、型に基づいた別名解析による最適化に設定できる仮定の内容を、-xalias_level オプションで指定されたレベルをもとに決定します。オプションで指定されたレベルは、コンパイル対象の翻訳単位に対して適用されます。第 1 のデフォルトは -xalias_level=any で、型に基づいた別名解析は行われないことを意味しています。値を指定しないで、つまり、 -xalias_level とだけ指定した場合は、デフォルトとして -xalias_level=layout が使用されます。

-xalias_level オプションで指定したレベルに定義されている、別名解析に関する仮定または制限事項のすべてに準拠しなかった場合は、プログラムの動作は未定義になるため注意が必要です。

-xalias_level=*list*

list には、次の表にあるいずれかのレベル名を指定します。

レベル名	意味
any	コンパイラはすべてのメモリー参照が別名であると仮定します。 -xalias_level=any では、型に基づいた別名解析は行いません。

レベル名	意味
basic	-xalias_level=basic オプションを使用すると、コンパイラは「異なる C の基本型を含むメモリー参照は互いに別名ではない」と仮定します。一方、「その他のすべての型に対する参照は互いに別名とみなせる」とも仮定します。コンパイラは「char * を使用する参照はどの型の参照とも別名とみなせる」と仮定します。 たとえば、-xalias_level=basic レベルでは、コンパイラは「型が int * であるポインタ変数は float 型のオブジェクトにはアクセスしない」と仮定します。つまり、コンパイラは「float * 型のポインタは、int * 型のポインタで参照されているメモリーを別名として参照することはない」と仮定した最適化を安全に行うことができます。
weak	-xalias_level=weak オプションを使用すると、コンパイラは「構造体ポインタは任意の型の構造体型を指すことができる」と仮定します。 コンパイル中のソースの中の式で参照されているか、コンパイル中のソースの外から参照されている型への任意の参照を含む構造体型または共用体型は、コンパイル中のソースの中の式の手前で宣言されなければなりません。 この制限に準拠するには、コンパイル中のソースの中の式で参照されている任意の型のオブジェクトを参照する型を含む、プログラムのヘッダーファイルをインクルードしてください。 -xalias_level=weak レベルでは、コンパイラは「異なる C の基本型を含むメモリー参照は、互いに別名ではない」と仮定します。また、「char * を使用する参照はどの型のメモリー参照とも別名とみなせる」とも仮定します。

レベル名	意味
layout	-xalias_level=layout オプションを使用すると、コンパイラは「同じ型を同じ順序で参照しているメモリー参照は別名とみなせる」と仮定します。 また、「メモリー中の同じ場所を参照しない型を持つ参照は、互いに別名とはみなさない」および「異なる struct 型を介した 2 つのメモリーアクセスは、構造体中の 1 つ目のメンバーがメモリー中の同じ場所を参照している場合は、互いに別名とみなす」とも仮定します。ただし、先頭のいくつかのメンバーがメモリー中で同じ場所を参照している 2 つの異なる構造体があった場合、共通の先頭部分より後にある struct フィールドにはポインタを使ってアクセスするべきではありません。これは、このレベルでは、コンパイラが上記のメモリー参照を互いに別名とみなさないためです。 -xalias_level=layout のレベルでは、コンパイラは「異なる C の基本型を含むメモリー参照は、互いに別名とはみなさない」と仮定します。また、「char * を使用している参照は、他のあらゆる型を持つ参照と別名とみなせる」とも仮定します。
strict	-xalias_level=strict オプションを使用すると、コンパイラは「構造体型や共用体型を含み、タグが削除されると同じになるメモリー参照は、互いに別名とみなせる」と仮定します。つまり、「タグを削除しても同じではない型を持つメモリー参照は別名とはみなさない」と仮定しています。 しかし、コンパイル中のソースの中の式で参照されているか、コンパイル中のソースの外から参照されている型への任意の参照を含む構造体型または共用体型は、コンパイル中のソースの中の式の手前で宣言されなければなりません。 この制限に準拠するには、コンパイル中のソースの中の式で参照されている任意の型のオブジェクトを参照する型を含む、プログラムのヘッダーファイルをインクルードしてください。 -xalias_level=strict レベルでは、コンパイラは「異なる C の基本型を含むメモリー参照は、互いに別名ではない」と仮定します。また、「char * を使用する参照はどの型の参照とも別名とみなせる」とも仮定します。
std	-xalias_level=std オプションを使用すると、コンパイラは「型とタグが同じでない限り別名とはみなせないが、char * を使用している参照は、型に関係なくどの参照とも別名とみなせる」と仮定します。この規則は 1999 年制定の ISO C 標準規格にあるポインタの間接参照に関する制約と同じです。この規則に則ったプログラムは移植性が高く、最適化することで大きなパフォーマンス向上が得られます。

レベル名	意味
strong	-xalias_level=strong オプションを使用すると、 std レベルと同じ内容に加えて、コンパイラは「char * 型のポインタは char * 型のオブジェクトにアクセスする場合にのみ使用される」と仮定します。また、「内部ポインタは存在しない」とも仮定します。内部ポインタの定義は「構造体のメンバーを指すポインタ」です。

プラグマ

型に基づいた別名解析をさらに細かく制御することで、より大きな効果が得られる場合には、以降で説明するプラグマを使用してください。このプラグマに翻訳単位中の個々の型またはポインタ変数に別名関係を指定すると、そのレベルはオプションで指定されたレベルより優先して適用されます。このプラグマは、翻訳単位全体には -xalias_level オプションのレベルが適用できるが、特定のポインタ変数が通常とは異なる方法で使用されているため、そのポインタ変数だけがオプションのレベルに準拠していない、といった場合に便利です。

注 - プラグマに指定する型または変数の名前は、そのプラグマの手前で宣言しなければなりません。宣言をしないと警告メッセージが発行され、プラグマは無視されます。プラグマの指定に当てはまる最初のメモリー参照の後でプラグマが指定されている場合は、プログラムの結果は未定義になります。

プラグマの定義では、次の用語を使用しています。

用語	意味
level	2 ページの「-xalias_level オプション」に列挙したレベル名のいずれかを指します。
type	次のいずれかを指します。 • `char`、`short`、`int`、`long`、`long long`、`float`、`double`、`long double` • `void` (すべてのポインタ型を指す) • `typedef` *name* (`typedef` 宣言で定義済みの型の名前が続く) • `struct` *name* (キーワード `struct` の後ろに構造体タグ名が続く) • `union` *name* (キーワード `union` の後ろに共用体タグ名が続く)
pointer_name	翻訳単位中のポインタ型を持つ任意の変数の名前

`#pragma alias_level` *level list*

level には、`any`、`basic`、`weak`、`layout`、`strict`、`std`、および `strong` の 7 つのレベルのうちの 1 つを指定します。*list* には型名またはポインタ名を指定できます。指定する型名またはポインタ名が複数ある場合は、コンマで区切って指定してください。 たとえば、次のように指定できます。

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

このプラグマは、指定された別名解析のレベルが、翻訳単位中で指定された型を持つすべてのメモリー参照に対して、または、指定されたすべてのポインタ変数が間接参照されている翻訳単位中のすべての間接参照に対して適用されます。

特定の間接参照に対して複数の別名解析レベルを指定した場合は、ポインタ名で指定されたレベルが (存在する場合は) その他のすべてのレベルより優先的に適用されます。同様に、型名で指定されたレベルが存在する場合は、そのレベルがオプションで

指定されたレベルより優先的に適用されます。次の例では、#pragma alias_level が any より高いレベルでプログラムがコンパイルされている場合、p には std レベルが適用されます。

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

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

このプラグマは「指定された型は互いに別名とみなす」と指定します。次の例では、コンパイラは「間接アクセス *pt は間接アクセス *pf の別名である」と指定します。

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

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

このプラグマは「指定されたポインタ変数を間接参照する時点で、間接参照されているポインタ値は指定されたその他のポインタ変数の間接参照を指すことできる」と指定します。このプラグマは、現在適用されている別名解析レベルの別名に関する仮定を無効にします。次の例では、コンパイラは「双方の型に関係なく、間接アクセス *p は間接アクセス *q の別名である」と仮定します。

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

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

このプラグマは「指定されたポインタ変数を間接参照する時点で、間接参照されているポインタ値は指定された変数に含まれているオブジェクトを指すことができる」と指定します。このプラグマは、現在適用されている別名解析レベルの別名に関する仮定を無効にします。次の例では、コンパイラは「間接アクセス *p は間接アクセス a、b、c の別名である」と仮定します。

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

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

このプラグマは「指定された型は互いに別名とはみなさない」と指定します。次の例では、コンパイラは「間接アクセス *p は間接アクセス *ps を別名とはみなさない」と仮定します。

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

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

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

このプラグマは、指定されたポインタ変数を間接参照する時点では、間接参照されているポインタ値は指定されたその他のポインタ変数と同じオブジェクトを指さない」と指定します。このプラグマは、現在適用されている別名解析レベルの別名に関する仮定を無効にします。次の例では、コンパイラは「双方の型に関係なく、間接アクセス *p は間接アクセス*q の別名ではない」と仮定します。

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

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

このプラグマは「指定されたポインタ変数を間接参照する時点で、間接参照されているポインタ値は指定された変数に含まれているオブジェクトを指さない」と指定します。このプラグマは、現在適用されている別名解析レベルの別名に関する仮定を無効にします。次の例では、コンパイラは「間接アクセス *p は間接アクセス a、b、c の別名ではない」と仮定します。

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

例題

この節では、-xalias_level オプションを使用した場合に、プログラム中のメモリー参照が受ける制約について例題を使って説明します。

例題 1

次のコード例をさまざまな別名解析レベルでコンパイルして、各型の別名解析の違いを比較してみましょう。

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

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

int *ip;
short *sp;
```

例題 1 を -xalias_level=any オプションでコンパイルすると、コンパイラは、以下の間接アクセスは互いに別名であると仮定します。

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

例題 1 を -xalias_level=basic オプションでコンパイルすると、コンパイラは、以下の間接アクセスは互いに別名であると仮定します。

*ip、*bp、fp->f1、fp->f4、bp->b1、bp->b2、bp->b3

さらに、*sp および fp->f2 と fp->f3、*sp と *fp は、互いに別名とみなせます。

ただし -xalias_level=basic オプションを指定すると、コンパイラは以下の内容を仮定します。

- *ip は *sp の別名ではない。
- *ip は fp->f2 および fp->f3 の別名ではない。
- *sp と fp->f1、fp->f4、bp->b1、bp->b2、bp->b3 は別名ではない。

これは、上記の間接アクセスのアクセス型が互いに異なる基本型であるためです。

例題 1 を -xalias_level=weak オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *ip は *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2、および bp->b3 と別名とみなせる。
- *sp は *fp、fp->f2、および fp->f3 の別名とみなせる。
- fp->f1 は bp->b1 の別名とみなせる。
- fp->f4 は bp->b3 の別名とみなせる。

コンパイラが fp->fp1 が bp->b2 の別名ではないと仮定します。これは、f1 が構造体の中で 0 オフセットのフィールドであるのに対して、b2 は構造体の中の 4 バイトオフセットのフィールドであるためです。同じ理由からコンパイラは、fp->f1 は bp->b3 の、fp->f4 は bp->b1 および bp->b2 の別名ではないと仮定します。

例題 1 を -xalias_level=layout オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *ip は *fp、*bp、fp->f1、fp->f4、bp->b1、bp->b2、および bp->b3 の別名とみなせる。
- *sp は *fp、fp->f2、および fp->f3 の別名とみなせる。
- fp->f1 は bp->b1 および *bp の別名とみなせる。

- *fp と *bp は、互いに別名とみなせる。

fp->f4 は bp->b3 の別名とはみなしません。これは、f4 と b3 が、foo および bar の先頭の共通部分であるフィールドに対応していないためです。

例題 1 を -xalias_level=strict オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *ip は *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2、および bp->b3 の別名とみなせる。
- *sp は *fp、fp->f2、および fp->f3 の別名とみなせる。

-xalias_level=strict を指定すると、コンパイラは「*fp、*bp、fp->f1、fp->f2、fp->f3、fp->f4、bp->b1、bp->b2、および bp->b3 は、互いに別名ではない」と仮定します。これは、フィールド名を無視した場合に foo と bar が同じではないためです。ただし、fp と fp->f1、および bp と bp->b1 は互いに別名とみなせます。

例題 1 を -xalias_level=std オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *ip は *fp、fp->f1、fp->f4、*bp、bp->b1、bp->b2、および bp->b3 の別名とみなせる。
- *sp は *fp、fp->f2、および fp->f3 の別名とみなせる。

ただし、コンパイラは「fp->f1 は bp->b1、bp->b2 または bp->b3 の別名ではない」と仮定します。これは、フィールド名を考慮した場合に foo と bar が同じではないためです。

例題 1 を -xalias_level=strong オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *ip は fp->f1、fp->f4、bp->b1、bp->b2、および bp->b3 の別名ではない。これは、*ip などのポインタは構造体の内部を指すべきではないという理由からです。

- *sp は fp->f1 または fp->f3 の別名ではない。
理由は上記と同じです。

- *ip は *fp、*bp、および *sp の別名ではない。
これは型が異なるためです。

- *sp は *fp、*bp、および *ip の別名ではない。
  これは型が異なるためです。

例題 2

次のコード例をさまざまな別名解析レベルでコンパイルして、各型の別名解析の違いを比較してみましょう。

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

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

例題 2 を -xalias_level=any オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 は、すべて互いに別名とみなせる。
  これは、-xalias_level=any のレベルでは、どのメモリーアクセスも互いに別名とみなせるためです。

例題 2 を -xalias_level=basic オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 は、すべて互いに別名とみなせる。ポインタ *fp および *bp を使用したフィールドアクセスは、互いに別名とみなせる。
  これは、構造体のすべての型が同じ基本型であるためです。

例題 2 を -xalias_level=weak オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *fp と *fp は、互いに別名とみなせる。
- fp->f1 は、bp->b1、*bp、および *fp と別名とみなせる。
- fp->f2 は、bp->b2、*bp、および *fp と別名とみなせる。

- fp->f3 は、bp->b3、*bp、および *fp と別名とみなせる。

ただし、-xalias_level=weak オプションを使用すると以下の制限事項も仮定されます。

- fp->f1 は、bp->b2 または bp->b3 の別名ではない。
  これは、f1 のオフセット (0 バイト) が、b2 のオフセット (4 バイトオフセット) および b3 のオフセット (8 バイトオフセット) と異なるためです。

- fp->f2 は、bp->b1 または bp->b3 の別名ではない。
  これは、f2 のオフセット (4 バイト) が b1 のオフセット (0 バイト) および b3 のオフセット (8 バイト) と異なるためです。

- fp->f3 は、bp->b1 または bp->b2 の別名ではない。
  これは、f3 のオフセット (8 バイト) が b1 のオフセット (0 バイト) および b2 のオフセット (4 バイト) と異なるためです。

例題 2 を -xalias_level=layout オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *fp と *bp は、互いに別名とみなせる。
- fp->f1 は、bp->b1、*bp、および *fp と別名とみなせる。
- fp->f2 は、bp->b2、*bp、および *fp と別名とみなせる。
- fp->f3 は、bp->b3、*bp、および *fp と別名とみなせる。

ただし、-xalias_level=layout オプションを使用すると以下の制限事項も仮定されます。

- fp->f1 は、 bp->b2 または bp->b3 の別名ではない。
  これは、フィールド f1 が、foo および bar の先頭の共通部分にあるフィールド b1 に対応するためです。

- fp->f2 は、bp->b1 または bp->b3 の別名ではない。
  これは、フィールド f2 が、foo および bar の先頭の共通部分にあるフィールド b2 に対応するためです。

- fp->f3 は、bp->b1 または bp->b2 の別名ではない。
  これは、フィールド f3 が、foo および bar の先頭の共通部分にあるフィールド b3 に対応するためです。

例題 2 を -xalias_level=strict オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- *fp と *bp は、互いに別名とみなせる。
- fp->f1 は、bp->b1、*bp、および *fp と別名とみなせる。
- fp->f2 は、bp->b2、*bp、および *fp と別名とみなせる。
- fp->f3 は、bp->b3、*bp、および *fp と別名とみなせる。

ただし、-xalias_level=strict オプションを使用すると以下の制限事項も仮定されます。

- fp->f1 は、bp->b2 または bp->b3 の別名ではない。
  これは、フィールド f1 が、foo および bar の先頭の共通部分にあるフィールド b1 に対応しているためです。

- fp->f2 は、bp->b1 または bp->b3 の別名ではない。
  これは、フィールド f2 が、foo および bar の先頭の共通部分にあるフィールド b2 に対応しているためです。

- fp->f3 は、bp->b1 または bp->b2 の別名ではない。
  これは、フィールド f3 が、foo および bar の先頭の共通部分にあるフィールド b3 に対応しているためです。

例題 2 を -xalias_level=std オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 は、互いに別名ではない。

例題 2 を -xalias_level=strong オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 は、互いに別名ではない。

例題 3

次のソースコードを例にして、特定の別名解析レベルでは内部ポインタを扱えないことについて説明します。内部ポインタの定義については 5 ページの「strong」の項目を参照してください。

```
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);
```

例題 3 の中の間接参照は、`weak`、`layout`、`strict`、または `std` のレベルでは別名とはみなせません。ポインタ代入 `bp=(struct bar*)(&fp->f2)` の後、次のように 2 つのメモリーアクセスがメモリー中の同じ場所を参照します。

- `fp->f2` および `bp->b2` は、メモリー中の同じ場所を参照する。
- `fp->f3` および `bp->b3` は、メモリー中の同じ場所を参照する。
- `fp->f4` および `bp->b4` は、メモリー中の同じ場所を参照する。

しかし、`weak`、`layout`、`strict`、および `std` のレベルでは、コンパイラは「`fp->f2` と `bp->b2` は別名ではない」と仮定します。これは、`b2` のオフセット(0 バイト) が `f2` のオフセット (4 バイト) と異なること、 および、`foo` と `bar` が互いに共通の先頭部分を持たないことが理由です。同じ理由から、コンパイラは「`bp->b3` は `fp->f3` の別名ではなく、`bp->b4` は `fp->f4` の別名ではない」と仮定します。

このように、ポインタ代入 `bp=(struct bar*)(&fp->f2)` によって、コンパイラの別名解析に関する仮定が正しくないものになってしまいます。そのため、最適化が正しく行われない可能性もあります。

この問題が発生しないように、次のようにコードの内容を変更してコンパイルしてみてください。

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

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

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

ポインタ代入 `bp=(struct bar*)(&fp->f2)` の後、次の 2 つのメモリーアクセスがメモリー中の同じ場所を参照します。

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

上のコードの変更部分でわかるように、`fp->f2` という式は `fp->fb.b2` とも表現できます。`fp->fb` の型は `bar` なので、`fp->f2` は `bar` のフィールド `b2` にアクセスします。さらに、`bp->b2` も `bar` のフィールド `b2` にアクセスします。従って、コンパイラは「`fp->f2` は `bp->b2` の別名である」と仮定します。同様に「`fp->f3` は `bp->b3` の、`fp->f4` は `bp->b4` の別名である」とも仮定します。結果として、コンパイラによる別名の仮定はポインタ代入による実際の別名と一致します。

例題 4

次のソースコードを例にして考えてみましょう。

```
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;
```

例題 4 を `-xalias_level=weak` オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- `fp->f1` は、`bp->b1`、`cp->c1`、`dp->d1`、`cp->cf.f1`、および `df->db.b1` を別名とみなせる。
- `fp->f2` は、`bp->b2`、`cp->cf.f1`、`dp->d2`、`cp->cf.f2`、`df->db.b2`、および `cp->c2` を別名とみなせる。
- `bp->b1` は、`fp->f1`、`cp->c1`、`dp->d1`、`cp->cf.f1`、および `df->db.b1` を別名とみなせる。
- `bp->b2` は、`fp->f2`、`cp->cf.f1`、`dp->d2`、`cp->cf.f1`、および `df->db.b2` を別名とみなせる。

`fp->f2` が `cp->c2` を別名とみなせるのは、`*dp` が `*cp` を、および `*fp` が `dp->db` を別名とみなせるためです。

- cp->c1 は、fp->f1、bp->b1、dp->d1、および dp->db.b1 を別名とみなせる。
- cp->cf.f1 は、fp->f1、fp->f2、bp->b1、bp->b2、dp->d2、および dp->d1 を別名とみなせる。

cp->cf.f1 は dp->db.b1 の別名ではありません。

- cp->cf.f2 は、fp->f2、bp->b2、dp->db.b1、および dp->d2 を別名とみなせる。
- cp->c2 は dp->db.b2 を別名とみなせる。

cp->c2 は dp->db.b1 の別名ではなく、cp->c2 は dp->d3 の別名ではありません。

オフセットについて考えれば、*dp が cp->cf の別名である場合にのみ、cp->c2 は db->db.b1 を別名とみなせます。しかし、*dp が cp->cf の別名である場合、dp->db.b1 は foo cf の末尾を越えた位置にある別名になってしまいます。これは、オブジェクトの制限で禁止されています。したがって、コンパイラは「cp->c2 は db->db.b1 の別名とはみなせない」と仮定します。

cp->c3 は dp->d3 を別名とみなせます。

cp->c3 が dp->db.b2 の別名でないことに注目してください。これは、間接参照に含まれる型を持つフィールドのオフセットが、2 つのメモリー参照間で異なり、重複していないためです。このため、コンパイラはこれらが別名ではないと仮定します。

- dp->d1 は fp->f1、bp->b1、および cp->c1 の別名とみなせる。
- dp->d2 は、fp->f2、bp->b2、および cp->cf.f1 の別名とみなせる。
- dp->db.b1 は、fp->f1、bp->b1、および cp->c1 の別名とみなせる。
- dp->db.b2 は、fp->f2、bp->b2、cp->c2、および cp->cf.f1 の別名とみなせる。
- dp->d3 は cp->c3 の別名とみなせる。

dp->d3 が cp->cf.f2 の別名でないことに注目してください。これは、間接参照に含まれる型を持つフィールドのオフセットが、2 つのメモリー参照間で異なり、重複していないためです。このため、コンパイラはこれらが別名ではないと仮定します。

例題 4 を -xalias_level=layout オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- fp->f1、bp->b1、cp->c1、および dp->d1 は、すべて互いに別名とみなせる。
- fp->f2、bp->b2、および dp->d2 は、すべて互いに別名とみなせる。

- fp->f1 は、cp->cf.f1 および dp->db.b1 の別名とみなせる。
- bp->b1 は、cp->cf.f1 および dp->db.b1 の別名とみなせる。
- fp->f2 は、cp->cf.f2 および dp->db.b2 の別名とみなせる。
- bp->b2 は、cp->cf.f2 および dp->db.b2 の別名とみなせる。

例題 4 を -xalias_level=strict オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- fp->f1 と bp->b1 は、互いに別名とみなせる。
- fp->f2 と bp->b2 は、互いに別名とみなせる。
- fp->f1 は、cp->cf.f1 および dp->db.b1 の別名とみなせる。
- bp->b1 は、cp->cf.f1 および dp->db.b1 の別名とみなせる。
- fp->f2 は、cp->cf.f2 および dp->db.b2 の別名とみなせる。
- bp->b2 は、cp->cf.f2 および dp->db.b2 の別名とみなせる。

例題 4 を -xalias_level=std オプションでコンパイルすると、コンパイラは以下の内容を仮定します。

- fp->f1 は cp->cf.f1 の別名とみなせる。
- bp->b1 は dp->db.b1 の別名とみなせる。
- fp->f2 は cp->cf.f2 の別名とみなせる。
- bp->b2 は dp->db.b2 の別名とみなせる。

例題 5

次のソースコードを例にして考えてみましょう。

```
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;
```

別名解析レベルごとのコンパイラの仮定内容は、次のようになります。

- 例題 5 を -xalias_level=weak オプションでコンパイルした場合、fp->f3 と bp->b2 は互いに別名とみなせる。
- 例題 5 を -xalias_level=layout オプションでコンパイルした場合、別名とみなせるフィールドはない。
- 例題 5 を -xalias_level=strict オプションでコンパイルした場合、fp->f3 と bp->b2 は互いに別名とみなせる。
- 例題 5 を -xalias_level=std オプションでコンパイルした場合、別名とみなせるフィールドはない。

例題 6

次のソースコードを例にして考えてみましょう。

```
struct bar;

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

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

別名解析レベルごとのコンパイラの仮定内容は、次のようになります。

- 例題 6 を -xalias_level=weak オプションでコンパイルした場合、fp->ffp と bp->bbp のみが互いに別名とみなせる。
- 例題 6 を -xalias_level=layout オプションでコンパイルした場合、fp->ffp と bp->bbp は互いに別名とみなせる。
- 例題 6 を -xalias_level=strict オプションでコンパイルした場合、別名とみなせるフィールドはない。
  これはタグを取り除いた後も、2 つの構造体の型が同じにならないためです。

- 例題 6 を -xalias_level=std オプションでコンパイルした場合、別名とみなせるフィールドはない。
  これは、2 つのフィールドの型とタグが同じでないためです。

例題 7

次のソースコードを例にして考えてみましょう。

```
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;
```

この例で使用されているプラグマは、コンパイラに「foo と bar は、互いに別名とみなせる」と指示します。これによって、コンパイラは次の別名関係を仮定します。

- fp->f1 は、bp->b1、bp->b2、および bp->b3 と別名とみなせる。
- fp->f2 は、bp->b1、bp->b2、および bp->b3 と別名とみなせる。