ARM コンパイラ

バージョン 6.02

armclang リファレンスガイド

ARM DUI0774CJ

ARM コンパイラ

armclang リファレンスガイド

リリース情報

ドキュメント履歴

発行	日付	機密保持ステータス	変更点
A	14 3 月 2014	非機密扱い	ARM コンパイラ v6.00 リリース
B	15 12 月 2014	非機密扱い	ARM コンパイラ v6.01 リリース
C	30 6 月 2015	非機密扱い	ARM コンパイラ v6.02 リリース

非機密著作権情報

LES-PRE-20349

補記

本書の一部の情報は、『IEEE 754 - 1985 IEEE Standard for Binary Floating-Point Arithmetic』に基づいています。記載されている方法による配置と使用から生じる責任または義務を IEEE では一切放棄しています。

機密保持ステータス

本書は非機密扱いであり、本書を使用、複製、および開示する権利は、ARM および ARM が本書を提供した当事者との間で締結した契約の条項に基づいたライセンスの制限により異なります。

無制限アクセスは、ARM 社内による分類です。

製品ステータス

本書の情報は最終版であり、開発済み製品に対応しています。

Web アドレス

http://www.jp.arm.com

目次
ARM コンパイラ armclang リファレンスガイド

図の一覧
ARM コンパイラ armclang リファレンスガイド

表の一覧
ARM コンパイラ armclang リファレンスガイド

序章

この前書きでは、次について紹介します。*ARM コンパイラ armclang リファレンスガイド*.

このドキュメントは、次で構成されています。

- *本書について*(10 ページ).

本書について

この『 ARM® コンパイラ armclang リファレンスガイド』では、ARM コンパイラである armclang に関する情報を提供します。armclang は、標準 C および標準 C++ のソースコードを ARM アーキテクチャベースプロセッサのマシンコードにコンパイルする、最適化 C および C++ コンパイラです。

本書の構成

本書は以下の章から構成されています。

第 章 1 コンパイラのコマンドラインオプション
armclang と共に使用する最も一般的なオプションの概要を示します。

第 章 2 コンパイラ固有のキーワードおよび演算子
C および C++ 標準の拡張機能であるコンパイラ固有のキーワードおよび演算子の概要を示します。

第 章 3 コンパイラ固有の関数、変数、および型属性
C および C++ 標準の拡張機能であるコンパイラ固有の関数、変数、および型属性の概要を示します。

第 章 4 コンパイラ固有のプラグマ
C および C++ 標準の拡張機能である ARM コンパイラ固有のプラグマを要約します。

第 章 5 その他のコンパイラ固有の機能
事前定義のマクロなど、C および C++ 標準の拡張機能であるコンパイラ固有の機能の概要を示します。

用語集

「ARM 用語集」は、ARM マニュアルで使用されている用語とその定義のリストです。一般に認められている意味と ARM での意味が異なる場合を除いて、「ARM 用語集」に業界標準の用語は含まれていません。

詳細については、「*ARM 用語集*」を参照して下さい。

表記規則

italic
重要用語、相互参照、引用箇所を示します。

bold
メニュー名などのユーザインタフェース要素を太字で記載しています。また、必要に応じて記述リスト内の重要箇所、ARM プロセッサの信号名、重要用語、および専門用語にも太字を使用しています。

`monospace`
コマンド、ファイル名、プログラム名、ソースコードなど、キーボードから入力可能なテキストを示しています。

`monospace`
コマンドまたはオプションに使用可能な略語を示しています。コマンド名またはオプション名をすべて入力する代わりに、下線部分の文字だけを入力することができます。

`monospace italic`
引数が特定の値で置き換えられる場合のモノスペーステキストの引数を示しています。

`monospace bold`
サンプルコード以外に使用される言語キーワードを示しています。

`<` および `>`
コードまたはコードの一部のアセンブラ構文で置換可能な項が使用されている場合に、その項を囲みます。以下はその一例です。

```
MRC p15, 0, &lt;Rd&gt;, &lt;CRn&gt;, &lt;CRm&gt;, &lt;Opcode_2&gt;
```

スモールキャピタル
「*ARM 用語集*」で定義されている専門的な意味を持つ用語について、本文中で使用されます。例えば、IMPLEMENTATION DEFINED、IMPLEMENTATION SPECIFIC、UNKNOWN、UNPREDICTABLE などです。

ご意見、ご感想

本製品に関するフィードバック

本製品についてのご意見やご提案がございましたら、以下の情報を添えて購入元までお寄せ下さい。
- 製品名
- 製品のリビジョンまたはバージョン
- 説明にはできるだけ多くの情報を含めて下さい。適宜、症状と診断手順も含めて下さい。

内容に関するフィードバック

内容に関するご意見につきましては、電子メールを *errata@arm.com* まで送信して下さい。その際には、以下の内容を記載して下さい。

- タイトル
- 文書番号（ARM DUI0774CJ）。
- 問題のあるページ番号
- 問題点の簡潔な説明

また、補足すべき点や改善すべき点についての全般的なご提案もお待ちしております。

——— **注** ———

ARM では、この PDF を Adobe Acrobat および Acrobat Reader でのみテストしており、その他の PDF リーダーを使用した場合の表示品質については、保証いたしかねます。

———————————

その他の情報

- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

第 1 章
コンパイラのコマンドラインオプション

armclang と共に使用する最も一般的なオプションの概要を示します。

armclang には、ARM にしかない多数のオプションやほとんどの Clang コマンドラインオプションなど、多くのコマンドラインオプションが装備されています。コマンドラインオプションに関する追加情報は、LLVM Compiler Infrastructure Project の Web サイト、*http://llvm.org* の Clang および LLVM のマニュアルでご覧いただけます。

注

以下の点に注意して下さい。

- 生成されたコードは 2 つの ARM® コンパイラリリース間で異なる場合があります。
- 機能のリリースについては、重大なコード生成の差異が存在する場合があります。

注

個々の ARM コンパイラツールのドキュメントのコマンドラインオプションの説明と関連情報で、ARM コンパイラ でサポートされている固有の機能がすべて記載されています。記載されていない ARM 固有の機能はすべて、サポート対象外のため、自己責任で使用して下さい。オープンソースの clang LLVM 機能を使用することもできますが、それらは ARM でサポートされていないため、自己責任で使用して下さい。サポートされていない機能を使用して生成されたコードについては、正しく動作することを必ず確認して下さい。

以下のセクションから構成されています。

1.1 -c

コンパイル手順を実行して、リンク手順は実行しないようにコンパイラに指示します。

使用法

-c オプションは、複数のソースファイルのプロジェクトでの使用を推奨します。

コンパイラは、入力ソースファイルのファイル拡張子を置換する .o ファイル拡張子が付いた各ソースファイルのオブジェクトファイルを 1 つ作成します。たとえば、以下は、オブジェクトファイル test1.o、test2.o、および test3.o を作成します。

```
armclang --target=aarch64-arm-none-eabi -c test1.c test2.c test3.c
```

注

-c オプションが付いたソースファイルを複数指定すると、-o オプションはエラーになります。以下に例を示します。

```
armclang --target=aarch64-arm-none-eabi -c test1.c test2.c -o test.o
armclang:エラー:複数の出力ファイルを生成する場合、-o を指定できません
```

1.2 -D

マクロ *name* を定義します。

構文

`-D` *name* `[(` *parm-list* `)][=` *def* `]`

各項目には以下の意味があります。

name
定義するマクロの名前です。

parm-list
コンマで区切られたマクロパラメータのリストを指定できます（省略可）。関数形式のマクロは、マクロパラメータリストをマクロ名に追加することで定義できます。
パラメータリストは括弧で囲む必要があります。複数のパラメータを指定する場合は、リストのコンマとパラメータ名の間にスペースを入れないで下さい。

—— **注** ——
UNIX システムでは、括弧をエスケープする必要があります。
——————————

=def
オプションのマクロ定義です。

=def が省略されている場合、 *name* を 1 と定義します。

トークンとして認識された文字をコマンドラインに含めるには、マクロ定義を二重引用符で囲みます。

使用法

`-D`*name* を指定したときの効果は、各ソースファイルの冒頭にテキスト `#define` *name* を記述したときの効果と同じです。

例

以下のこのオプションを指定します。

```
-DMAX(X,Y)="((X > Y) ?X :Y)"
```

は、以下のマクロの定義と同等です。

```
#define MAX(X, Y) ((X > Y) ?X :Y)
```

を定義することと同じ意味です。

1.3 -E

プロセッサの手順のみを実行します。

デフォルトでは、プリプロセッサからの出力は標準出力ストリームに送信され、標準の UNIX または MS-DOS の指定方法を使用してファイルに転送できます。

`-o` オプションを使用して、前処理済み出力用のファイルを指定できます。

デフォルトでは、コメントは出力から除外されます。`-C` オプションを使用して、前処理済み出力にコメントを保持します。

インターリーブされたマクロ定義とプリプロセッサ出力を生成するには、`-E -dD` を使用します。

例

```
armclang --target=aarch64-arm-none-eabi -E -dD source.c > raw.c
```

1.4 -e

イメージ固有の初期エントリポイントを指定します。

`armclang` はこのオプションを `--entry` に変換し、`armlink` に渡します。

`--entry` リンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

関連情報

ARM コンパイラツールチェーンリンカリファレンス.

1.5 -fbare-metal-pie

位置非依存コードを生成します。

このオプションを使用すると、コンパイラは、リンク手順を実行するときに `--bare_metal_pie` オプションを使用して `armlink` を呼び出します。

——— 注 ———

AArch64 状態ではサポートされていません。

———————————

関連情報

ベアメタル位置非依存実行可能ファイル.

--fpic armlink オプション.

--pie armlink オプション.

--bare_metal_pie armlink オプション.

--ref_pre_init armlink オプション.

1.6 -fcommon, -fno-common

仮定義としてゼロで初期化された共通の定義を生成します。

仮定義は記憶域クラスおよびイニシャライザのない変数の宣言です。

-fno-common オプションは、仮定義としてゼロで初期化された個々の定義を生成します。これらのゼロで初期化された定義は、生成されたオブジェクトの ZI セクションに配置されます。異なるファイルに複数の定義が存在する場合、個々の定義が互いに衝突するため、L6200E:シンボルの多重定義リンカエラーが発生します。

-fcommon オプションは、ゼロで初期化された定義を共通ブロックに配置します。この共通定義は特定のセクションまたはオブジェクトに関連付けられていないため、リンク時に複数の定義が 1 つの定義として解決されます。

デフォルト

デフォルトは -fcommon です。

1.7 -ffast-math

このオプションを設定することで、強力な浮動小数点の最適化を実行するようコンパイラに指示します。

その結果、ISO C および C++ 標準に対して完全には準拠しない動作となります。ただし、数値的に頑健な浮動小数点プログラムは正しく動作します。

1.8 @file

ファイルからコンパイラオプションのリストを読み出します。

構文

@file

file は、コマンドラインでインクルードされる armclang オプションを含むファイルの名前です。

使用法

指定ファイルのオプションは、*@file* オプションの代わりに挿入されます。

空白または新しい行を使用して、ファイルのオプションを区切ります。単一または二重引用符で文字列を囲み、1 ワードとして処理します。

コマンドラインに *@file* オプションを複数指定して、複数のファイルからのオプションを含めることができます。ファイルには、*@file* オプションをさらに含めることができます。

@file オプションが存在しないファイルまたは循環依存を指定している場合、armclang はエラーを表示して終了します。

例

次の内容で、ファイル options.txt を検討してください。

```
"-I../my libs/"
--target=aarch64-arm-none-eabi -mcpu=cortex-a57
```

以下のコマンドラインを使用して、ソースファイル main.c をコンパイルします。

```
armclang @options.txt main.c
```

このコマンドは以下に相当します。

```
armclang -I"../my libs/" --target=aarch64-arm-none-eabi -mcpu=cortex-a57 main.c
```

1.9 -fno-inline-functions

関数のインライン展開を無効にすると、デバッグ機能を改善できます。

最適化レベル `-O2` 以上で、オプション `-fno-inline-functions` が選択されている場合、コンパイラは関数を自動的にインライン展開することはありません。

関連概念

5.2 インライン関数(5-117 ページ).

関連参照

1.32 -O(1-47 ページ).

1.10 -flto

リンク時最適化を有効にし、ELF オブジェクトファイルではなくリンク時最適化を行うビットコードファイルを出力します。

ビットコードファイルは主にリンク時最適化に使用します。リンク時最適化の詳細については、*『ソフトウェア開発ガイド』*の*「リンク時最適化を使用したモジュール間の最適化」*を参照して下さい。

注

ARM コンパイラ は、32 ビットの Red Hat Enterprise Linux プラットフォームでのリンク時最適化をサポートしていません。

使用法

コンパイラは、`.o` ファイル拡張子で入力ソースファイルのファイル拡張子を置き換えて、各ソースファイルに 1 つのビットコードファイルを作成します。

`-c` オプションが指定されていない場合、`-flto` オプションは、`--lto` オプションを `armlink` に渡し、リンク時最適化を有効にします。

関連参照

1.1 -c(1-14 ページ).

関連情報

リンク時最適化を使用したモジュール間の最適化.

--lto armlink オプション.

1.11 -fno-exceptions

C++ 例外をサポートするのに必要なコードの生成を抑制します。

使用法

`-fno-exceptions` オプションは、C++ 標準ライブラリ libc++ と共に使用できますが、以下に注意してください。

- ARM コンパイラ 6 には、`-fno-exceptions` でビルドする libc++ のバリアントは含まれていません。このため、ビルド済みの libc++ オブジェクトから例外がスローされることがあります。
- ヘッダファイルに実装される libc++ の一部は、`-fno-exceptions` でコンパイルされることがあります。`-fno-exceptions` が使用されている場合、libc++ は、例外をスローする代わりに assert マクロを呼び出します。

  注

  マクロ `NDEBUG` が定義されている場合、これらの assert マクロは削除されます。

try、catch、または throw を使用すると、エラーメッセージになります。別のオブジェクトからの C++ 例外が、`-fno-exceptions` でビルドされたコードに伝播する場合、プログラムは終了します。

Rogue Wave C++ ライブラリ

`armclang` と Rogue Wave C++ ライブラリは、スキームをサポートする互換性のない、異なる例外を使用します。このため、`-stdlib=legacy_cpplib` を使用する際は、`-fno-exceptions` を指定する必要があります。

関連参照

1.36 -stdlib（1-51 ページ）.

関連情報

標準 C++ ライブラリの実装定義.

1.12 -fshort-enums, -fno-short-enums

コンパイラが列挙型のサイズを、すべての列挙子の値を保持できる最小のデータ型に設定できるようにします。

-fshort-enums オプションはメモリ使用量を改善できますが、局所的なメモリアクセスはレジスタ幅全体のアクセスよりも効率が低下する場合があるため、パフォーマンスが低下することがあります。

注

ライブラリを含め、リンクされたすべてのオブジェクトが同じ選択をする必要があります。-fshort-enums を使用してコンパイルしたオブジェクトファイルは、-fshort-enums を使用しないでコンパイルした別のオブジェクトファイルにリンクできません。

注

AArch64 では、-fshort-enums オプションはサポートされていません。*ARM® 64 ビットアーキテクチャ向けプロシージャコール標準*では、列挙型のサイズは 32 ビット以上である必要があります。-fshort-enums オプションが AArch64 ターゲットに指定されている場合は無視されます。

デフォルト

デフォルトは -fno-short-enums です。つまり、列挙型のサイズは列挙値のサイズにかかわらず 32 ビット以上です。

例

この例では、4 つの列挙型のサイズ（8 ビット、16 ビット、32 ビット、64 ビット整数）を示しています。

```
#include <stdio.h>

// 最大値は 8 ビット整数
enum int8Enum  {int8Val1 =0x01, int8Val2 =0x02, int8Val3 =0xF1 };

// 最大値は 16 ビット整数
enum int16Enum {int16Val1=0x01, int16Val2=0x02, int16Val3=0xFFF1 };

// 最大値は 32 ビット整数
enum int32Enum {int32Val1=0x01, int32Val2=0x02, int32Val3=0xFFFFFFF1 };

// 最大値は 64 ビット整数
enum int64Enum {int64Val1=0x01, int64Val2=0x02, int64Val3=0xFFFFFFFFFFFFFFF1 };

int main(void)
{
  printf("size of int8Enum  is %zd\n", sizeof (enum int8Enum));
  printf("size of int16Enum is %zd\n", sizeof (enum int16Enum));
  printf("size of int32Enum is %zd\n", sizeof (enum int32Enum));
  printf("size of int64Enum is %zd\n", sizeof (enum int64Enum));
}
```

-fshort-enums オプションを使用しないでコンパイルした場合、64 ビット（8 バイト）を必要とする int64Enum を除き、すべての列挙型は 32 ビット（4 バイト）です。

```
armclang --target=arm-arm-eabi-none -march=armv8-a enum_test.cpp

size of int8Enum  is 4
size of int16Enum is 4
size of int32Enum is 4
size of int64Enum is 8
```

-fshort-enums オプションを使用してコンパイルした場合、各列挙型は最大の列挙値を保持できる最小サイズになります。

```
armclang -fshort-enums --target=arm-arm-eabi-none -march=armv8-a enum_test.cpp

size of int8Enum  is 1
size of int16Enum is 2
```

```
size of int32Enum is 4
size of int64Enum is 8
```

注

ISO C は列挙値を `int` の範囲に制限します。デフォルトでは、`armclang` は大きすぎる列挙値について警告を表示しませんが、`-Wpedantic` を使用すると警告が表示されます。

関連情報

ARM 64 ビットアーキテクチャ向けプロシージャコール標準(AArch64).

1.13 -fshort-wchar, -fno-short-wchar

wchar_t のサイズを 2 バイトに設定します。

-fshort-wchar オプションはメモリ使用量を改善できますが、局所的なメモリアクセスはレジスタ幅全体のアクセスよりも効率が低下する場合があるため、パフォーマンスが低下することがあります。

注

ライブラリを含め、リンクされたすべてのオブジェクトが同じ wchar_t サイズを使用する必要があります。-fshort-wchar を使用してコンパイルしたオブジェクトファイルは、-fshort-wchar を使用しないでコンパイルした別のオブジェクトファイルにリンクできません。

デフォルト

デフォルトは -fno-short-wchar です。つまり、wchar_t のデフォルトのサイズは 4 バイトです。

例

この例では、wchar_t 型のサイズを示します。

```
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  printf("size of wchar_t is %zd\n", sizeof (wchar_t));
  return 0;
}
```

-fshort-wchar オプションを使用しないでコンパイルした場合、wchar_t のサイズは 4 バイトです。

```
armclang --target=aarch64-arm-none-eabi wchar_test.c

size of wchar_t is 4
```

-fshort-wchar オプションを使用してコンパイルした場合、wchar_t のサイズは 2 バイトです。

```
armclang -fshort-wchar --target=aarch64-arm-none-eabi wchar_test.c

size of wchar_t is 2
```

1.14 -fvectorize, -fno-vectorize

最適化レベル -O1 以上で C または C++ コードから直接、Advanced SIMD ベクトル命令の生成を有効および無効にします。

——— 注 ———

-fvectorize オプションは、AArch64 ステートではサポートされません。コンパイラは、AArch64 ステートターゲットの自動ベクトル化を実行することはありません。

———————————

デフォルト

デフォルトは使用する最適化レベルによって異なります。

最適化レベル -O0（デフォルトの最適化レベル）armclang は、自動ベクトル化を実行することはありません。-fvectorize および -fno-vectorize オプションは無視されます。

最適化レベル -O1 では、デフォルトは -fno-vectorize です。自動ベクトル化を有効にするには、-fvectorize を使用します。

最適化レベル -O2 以上では、デフォルトは -fvectorize です。自動ベクトル化を無効にするには、-fno-vectorize を使用します。

例

この例では、最適化レベル -O1 で自動ベクトル化を有効にします。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -fvectorize -O1 -c file.c
```

関連参照

1.1 -c（1-14 ページ）.
1.32 -O（1-47 ページ）.

1.15 -g, -gdwarf-2, -gdwarf-3, -gdwarf-4,

ソースレベルのデバッグにデバッグテーブルを追加します。

構文

`-g`

`-gdwarf-` *version*

各項目には以下の意味があります。

version
生成する DWARF 形式です。有効な値は、2、3、および 4 です。

`-g` オプションは、`-gdwarf-4` と同じ意味です。

使用法

コンパイラは、指定された DWARF 標準と互換性のあるデバッグ情報を生成します。

互換デバッガを使用して、イメージをロード、実行、およびデバッグします。例えば、ARM DS-5 デバッガは DWARF 4 と互換性があります。`-g` または `-gdwarf-4` オプションを使用してコンパイルし、ARM DS-5 デバッガを使用してデバッグできます。

従来のツールやサードパーティ製のツールは DWARF 4 デバッグ情報をサポートしていない場合があります。この場合は、`-gdwarf-2` または `-gdwarf-3` オプションを使用して、必要な DWARF 適合性のレベルを指定できます。

DWARF 4 仕様では、以前のバージョンの DWARF では使用できない言語機能をサポートしているため、`-gdwarf-2` および `-gdwarf-3` オプションは下位互換性を保つためだけに使用する必要があります。

デフォルト

デフォルトでは、`armclang` はデバッグ情報を生成しません。

サンプル

複数のオプションを指定した場合、最後に指定されたオプションが優先されます。以下に例を示します。

- `-gdwarf-3 -gdwarf-2` では、`-gdwarf-2` が `-gdwarf-3` をオーバーライドするため、DWARF 2 デバッグが生成されます。
- `-g -gdwarf-2` では、`-gdwarf-2` が `-g`（`-gdwarf-4` と同じ意味）をオーバーライドするため、DWARF 2 デバッグが生成されます。
- `-gdwarf-2 -g` では、`-g`（`-gdwarf-4` と同じ意味）が `-gdwarf-2` をオーバーライドするため、DWARF 4 デバッグが生成されます。

1.16 -I

指定したディレクトリを、インクルードされるファイルを見つけるために検索する場所のリストに追加します。

複数のディレクトリを指定した場合、それらのディレクトリは `-I` オプションに指定した順序で検索されます。

構文

`-I` *dir*

各項目には以下の意味があります。

dir
は、インクルードされるファイルを検索するディレクトリです。

複数の `-I` オプションを使用して、複数の検索ディレクトリを指定します。

1.17 -L

リンカがユーザライブラリの検索に使用するパスのリストを指定します。

構文

`-L dir [, dir ,...]`

各項目には以下の意味があります。

dir`[,`*dir*`,...]`
ユーザーライブラリの検索対象となるディレクトリのコンマ区切りのリストです。

ただし、少なくとも 1 つのディレクトリが指定されている必要があります。

複数のディレクトリを指定する場合は、リストのコンマとディレクトリ名の間にスペースを入れないで下さい。

`armclang` はこのオプションを `--userlibpath` に変換し、`armlink` に渡します。

`--userlibpath` リンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

注

`-L` オプションは、`-c` オプションと組み合わせて使用した場合（つまりリンク時以外）は機能しません。

関連情報

ARM コンパイラツールチェーンリンカリファレンス.

1.18 -l

指定されたライブラリを検索対象ライブラリの一覧に追加します。

構文

`-l` *name*

ここで、 *name* は、ライブラリの名前です。

`armclang` はこのオプションを `--library` に変換し、`armlink` に渡します。

`--library` リンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

注

`-l` オプションは、`-c` オプションと組み合わせて使用した場合（つまりリンク時以外）は機能しません。

関連情報

ARM コンパイラツールチェーンリンカリファレンス.

1.19 -M

make ユーティリティによる使用に適したメイクファイル依存関係規則のリストを生成します。

このオプションを指定すると、コンパイルのプリプロセッサ処理のみが実行されます。デフォルトでは、標準出力ストリームに出力されます。

複数のソースファイルを指定した場合は、1 つの依存関係ファイルが作成されます。

—— 注 ——

-MT オプションにより、依存関係規則のターゲット名をオーバーライドできます。

—— 注 ——

-MD オプションにより、メイクファイル依存関係規則の生成に加えて、ソースファイルのコンパイルを行えます。

例

出力は、標準の UNIX または MS-DOS の指定方法、-o オプション、または -MF オプションを使用してファイルに転送できます。以下に例を示します。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c &gt; deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -o deps.mk
armclang --target=arm-arm-none-eabi -march=armv8-a -M source.c -MF deps.mk
```

関連参照

1.31 -o（1-46 ページ）.
1.20 -MD（1-34 ページ）.
1.21 -MF（1-35 ページ）.
1.22 -MT（1-36 ページ）.

1.20 -MD

ソースファイルをコンパイラして、make ユーティリティでの使用に適したメイクファイル用の依存関係規則のリストを生成します。

コンパイラは、.d 接尾文字を使用して、各ソースファイルのメイクファイル依存関係ファイルを作成します。

例

以下の例では、メイクファイル依存関係リスト test1.d および test2.d を作成して、デフォルト名が a.out のイメージにソースファイルをコンパイルします。

```
armclang --target=arm-arm-none-eabi -march=armv8-a -MD test1.c test2.c
```

関連参照

1.19 -M（1-33 ページ）.
1.21 -MF（1-35 ページ）.
1.22 -MT（1-36 ページ）.

1.21 -MF

-M および -MD オプションによって生成されるメイクファイル依存関係の規則のファイル名を指定します。

構文

-MF *file*

各項目には以下の意味があります。

file
メイクファイル依存関係の規則のファイル名を指定します。

注

-MF オプションは、-M または -MD オプションのいずれかと組み合わせたときにのみ機能します。

armclang -M は、デフォルトで標準出力ストリームに出力を送信します。それに対し、-MF オプションは、指定されたファイル名に出力を送信します。

-M を使用して複数のソースファイルを指定した場合、すべてのソースファイルのメイクファイル依存関係の規則は連結されます。また、-MF を指定した場合、連結された依存関係の規則は指定されたファイルに保存されます。

armclang -MD は、デフォルトでソースファイルと同名のファイルに出力を送信しますが、.d 接尾文字を付けます。それに対し、-MF オプションは、指定されたファイル名に出力を送信します。armclang -MD -MF では 1 つのソースファイルを使用して下さい。

サンプル

この例では、ソースをコンパイルせずに、メイクファイル依存関係の規則を標準出力に送信します。

```
armclang --target=aarch64-arm-none-eabi -M source.c
```

この例では、ソースをコンパイルせずに、メイクファイル依存関係の規則を deps.mk に保存します。

```
armclang --target=aarch64-arm-none-eabi -M source.c -MF deps.mk
```

この例では、ソースをコンパイルせずに、複数のソースファイルの連結されたメイクファイル依存関係の規則を標準出力に送信します。

```
armclang --target=aarch64-arm-none-eabi -M source1.c source2.c source3.c
```

この例では、ソースをコンパイルせずに、複数のソースファイルの連結されたメイクファイル依存関係の規則を deps.mk に送信します。

```
armclang --target=aarch64-arm-none-eabi -M source1.c source2.c source3.c -MF deps.mk
```

この例では、ソースファイルをコンパイルし、メイクファイル依存関係の規則を source.d に保存します（デフォルトのファイル命名規則を使用）。

```
armclang --target=aarch64-arm-none-eabi -MD source.c
```

この例では、ソースをコンパイルし、メイクファイル依存関係の規則を deps.mk に保存します。

```
armclang --target=aarch64-arm-none-eabi -MD source.c -MF deps.mk
```

関連参照

1.19 -M（1-33 ページ）.
1.20 -MD（1-34 ページ）.
1.22 -MT（1-36 ページ）.

1.22 -MT

-M が生成したメイクファイル依存関係規則のターゲットを変更します。

注

-MT オプションは、-M または -MD オプションと共に使用する場合にのみ機能します。

デフォルトでは、armclang -M は、以下のソースファイル名に基づいて、メイクファイル依存関係規則を作成します。

```
armclang --target=aarch64-arm-none-eabi -M test.c
test.o: test.c header.h
```

-MT オプションは、メイクファイル依存関係規則のターゲットの名前を変更します。

```
armclang --target=aarch64-arm-none-eabi -M test.c -MT foo
foo: test.c header.h
```

このオプションを指定すると、コンパイルのプリプロセッサ処理のみが実行されます。デフォルトでは、標準出力ストリームに出力されます。

複数のソースファイルを指定する場合、-MT オプションは、すべての依存関係規則のターゲットの名前を変更します。

```
armclang --target=aarch64-arm-none-eabi -M test1.c test2.c -MT foo
foo: test1.c header.h
foo: test2.c header.h
```

複数の -MT オプションを指定すると、以下の各規則のターゲットが複数個作成されます。

```
armclang --target=aarch64-arm-none-eabi -M test1.c test2.c -MT foo -MT bar
foo bar: test1.c header.h
foo bar: test2.c header.h
```

関連参照

1.19 -M(1-33 ページ).
1.20 -MD(1-34 ページ).
1.21 -MF(1-35 ページ).

1.23 -march

アーキテクチャプロファイルをターゲットにすると、そのアーキテクチャを持つどのプロセッサ上でも動作する汎用コードが生成されます。

構文

`-march=` *name*

各項目には以下の意味があります。

name

アーキテクチャを指定します。

以下は、有効な `-march` 値です。

`armv8-a`

ARMv8-A アーキテクチャプロファイル。`--target=aarch64-arm-none-eabi` および `--target=arm-arm-none-eabi` の両方で有効。

`armv7-a`

ARMv7-A アーキテクチャプロファイル。`--target=arm-arm-none-eabi` でのみ有効です。

デフォルト

AArch64 ターゲット(`--target=aarch64-arm-none-eabi`)の場合、`-mcpu`. を使用して、特定のプロセッサをターゲットにしない限り、デフォルトで `-march=armv8-a` になり、AArch64 状態で ARMv8-A の汎用コードが生成されます。

AArch32 ターゲット(`--target=arm-arm-none-eabi`)の場合、デフォルトはありません。`-march`(アーキテクチャをターゲットにする場合)または `-mcpu`(プロセッサをターゲットにする場合)のいずれかを指定する必要があります。

1.24 -marm

A32 または ARM 命令セットをターゲットとするようにコンパイラに要求します。

以下のようにアーキテクチャによって異なる命令セットがサポートされています。

- AArch64 状態の ARMv8-A プロセッサは、A64 命令を実行します。
- AArch32 状態の ARMv8-A プロセッサは、A32 または T32 命令を実行できます。
- ARMv7-A プロセッサは、ARM または Thumb 命令を実行できます。

`-marm` オプションは、A32（ARMv8-A AArch32 状態）または ARM（ARMv7-A）命令セットをターゲットとします。これは、`arm-arm-none-eabi` ターゲットのデフォルトです。

注

`-marm` オプションは、`--target=aarch64-arm-none-eabi` などの AArch64 ターゲットでは無効です。AArch64 ターゲットでは、コンパイラは `-marm` オプションを無視し、警告を生成します。

デフォルト

ARMv8-A AArch32 および ARMv7-A ターゲットのデフォルトは `-marm` です。

関連参照

1.30 -mthumb（1-45 ページ）.
1.37 --target（1-52 ページ）.
1.26 -mcpu（1-40 ページ）.

関連情報

ターゲットアーキテクチャ、プロセッサ、および命令セットの指定.

1.25 -mbig-endian

バイトインバリアントビッグエンディアン（BE-8）データを使用して、ARM プロセッサに適したコードを生成します。

デフォルト

デフォルトは `-mlittle-endian` です。

関連参照

1.29 -mlittle-endian（1-44 ページ）.

1.26 -mcpu

このオプションを使用すると、特定の ARM プロセッサを対象としたコードが生成されます。

構文

`-mcpu=` *name*

`-mcpu=` *name*`[+[no]`*feature*`]*`（AArch64 ターゲットのみ）

各項目には以下の意味があります。

name

プロセッサを指定します。

以下は、`--target=aarch64-arm-none-eabi` と `--target=arm-arm-none-eabi` で有効な `-mcpu` 値です。

- `cortex-a53`
- `cortex-a57`
- `cortex-a72`

以下は、`--target=arm-arm-none-eabi` のみで有効な `-mcpu` 値です。

- `cortex-a5`
- `cortex-a7`
- `cortex-a8`
- `cortex-a9`
- `cortex-a12`
- `cortex-a15`
- `cortex-a17`

feature

以下のいずれかのオプションのアーキテクチャ機能（AArch64 ターゲットのみ）を有効または無効にします。

- `crc` - CRC 命令を有効にします。
- `crypto` - 暗号化拡張機能を有効にします。
- `fp` - 浮動小数点拡張を有効にします。
- `simd` - NEON advanced SIMD 拡張を有効にします。

注

*名前*と*機能*では大文字と小文字が区別されます。

使用法

AArch64 ターゲットの場合のみ、`-mcpu` オプションを使用して特定のアーキテクチャ機能を有効または無効にします。

機能を無効にするには、`cortex-a57+nocrypto` などのように、接頭文字 `no` を使用します。

複数の機能を有効または無効にするには、複数の機能の修飾子をつなぎます。例えば、CRC 命令を有効にし、他のすべての拡張機能を無効にするには、次のようにします。

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nocrypto+nofp+nosimd+crc
```

`-mcpu` を使用して競合する機能の修飾子を指定した場合、最も右側にある機能が使用されます。例えば、以下のコマンドは浮動小数点拡張を有効にします。

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57+nofp+fp
```

`-mcpu=name+nofp+nosimd` オプションを使用して、AArch64 ターゲットが浮動小数点命令または浮動小数点レジスタを使用しないようにすることができます。このモードで、その後に浮動小数点データ型を使用することはできません。

デフォルト

AArch64 ターゲット(`--target=aarch64-arm-none-eabi`)の場合、コンパイラは AArch64 状態で ARMv8-A アーキテクチャの汎用コードをデフォルトで生成します。

AArch32 ターゲット(`--target=arm-arm-none-eabi`)の場合、デフォルトはありません。`-march`(アーキテクチャをターゲットにする場合)または `-mcpu`(プロセッサをターゲットにする場合)のいずれかを指定する必要があります。

例

Cortex® -A57 プロセッサの AArch64 状態をターゲットにする場合は以下のとおりです。

```
armclang --target=aarch64-arm-none-eabi -mcpu=cortex-a57 test.c
```

Cortex -A53 プロセッサの AArch32 状態をターゲットにする場合は以下の A32 命令で生成します。

```
armclang --target=arm-arm-none-eabi -mcpu=cortex-a53 -marm test.c
```

関連参照

1.28 -mfpu(1-43 ページ).
1.24 -marm(1-38 ページ).
1.30 -mthumb(1-45 ページ).
1.37 --target(1-52 ページ).
1.28 -mfpu(1-43 ページ).
1.37 --target(1-52 ページ).

関連情報

ターゲットアーキテクチャ、プロセッサ、および命令セットの指定.
浮動小数点命令とレジスタの使用の回避.

1.27 -mfloat-abi

浮動小数点演算にハードウェア命令を使用するか、ソフトウェアライブラリ関数を使用するかを指定し、浮動小数点パラメータおよび戻り値を渡すために使用するレジスタを指定します。

構文

`-mfloat-abi=` *value*

value には以下のいずれかを指定できます。

`soft`
: 浮動小数点演算およびソフトウェア浮動小数点リンケージ向けのソフトウェアライブラリ関数。

`softfp`
: ハードウェア浮動小数点命令とソフトウェア浮動小数点リンケージ。

`hard`
: ハードウェア浮動小数点命令とハードウェア浮動小数点リンケージ。

注

`-mfloat-abi` オプションは ARMv8 AArch64 ターゲットでは無効です。AArch64 ターゲットは、ハードウェア浮動小数点命令およびハードウェア浮動小数点リンケージを使用します。ただし、`-mcpu=`*name* `+nofp+nosimd` オプションを使用して、AArch64 ターゲットが浮動小数点命令または浮動小数点レジスタを使用しないようにすることができます。このモードで、その後に浮動小数点データ型を使用することはできません。

デフォルト

`--target=arm-arm-none-eabi` のデフォルトは、`softfp` です。

関連参照

1.28 -mfpu（1-43 ページ）.

1.28 -mfpu

ターゲットの FPU アーキテクチャを指定します。これはターゲットで使用できる浮動小数点ハードウェアです。

構文

`-mfpu=` *name*

name には、次のいずれかを指定します。

`vfpv3`
: ARMv7 VFPv3 浮動小数点拡張を有効にします。Advanced SIMD 拡張を無効にします。

`neon-vfpv3`
: ARMv7 VFPv3 浮動小数点拡張および Advanced SIMD 拡張を有効にします。

`vfpv4`
: ARMv7 VFPv4 浮動小数点拡張を有効にします。Advanced SIMD 拡張を無効にします。

`neon-vfpv4`
: ARMv7 VFPv4 浮動小数点拡張および Advanced SIMD 拡張を有効にします。

`fp-armv8`
: ARMv8 浮動小数点拡張を有効にします。暗号化拡張機能および Advanced SIMD 拡張を無効にします。

`neon-fp-armv8`
: ARMv8 浮動小数点拡張および Advanced SIMD 拡張を有効にします。暗号化拡張機能を無効にします。

`crypto-neon-fp-armv8`
: ARMv8 浮動小数点拡張、暗号化拡張機能、および Advanced SIMD 拡張を有効にします。

`-mfpu` オプションは、ターゲットのアーキテクチャにより暗示されたデフォルトの FPU オプションをオーバーライドします。

注

- `-mfpu` オプションは、`aarch64-arm-none-eabi` などの AArch64 ターゲットでは無視されます。`aarch64-arm-none-eabi` ターゲットのデフォルト FPU をオーバーライドするには、`-mcpu` オプションを使用します。例えば、`aarch64-arm-none-eabi` ターゲットが浮動小数点命令または浮動小数点レジスタを使用しないようにするには、`-mcpu=`*name*`+nofp+nosimd` オプションを使用します。このモードで、その後に浮動小数点データ型を使用することはできません。
- `arm-arm-none-eabi` のデフォルトである `-mfloat-abi=soft` の場合、`-mfpu` オプションも無視されます。
- ARMv7 では、Advanced SIMD 拡張機能は、NEON Advanced SIMD 拡張機能と呼ばれていました。

デフォルト

デフォルトの FPU オプションはターゲットアーキテクチャによって異なります。

関連参照

1.26 -mcpu（1-40 ページ）.
1.27 -mfloat-abi（1-42 ページ）.
1.26 -mcpu（1-40 ページ）.
1.37 --target（1-52 ページ）.

関連情報

ターゲットアーキテクチャ、プロセッサ、および命令セットの指定.
浮動小数点命令とレジスタの使用の回避.

1.29 -mlittle-endian

リトルエンディアンデータを使用して、ARM プロセッサに適したコードを生成します。

デフォルト

デフォルトは `-mlittle-endian` です。

関連参照

1.25 -mbig-endian（1-39 ページ）.

1.30 -mthumb

T32 または Thumb® 命令セットをターゲットとするようにコンパイラに要求します。

以下のようにアーキテクチャによって異なる命令セットがサポートされています。

- AArch64 状態の ARMv8-A プロセッサは、A64 命令を実行します。
- AArch32 状態の ARMv8-A プロセッサは、A32 または T32 命令を実行できます。
- ARMv7-A プロセッサは、ARM または Thumb 命令を実行できます。

`-mthumb` オプションは、T32（ARMv8-A AArch32 状態）または Thumb（ARMv7-A）命令セットをターゲットとします。

注

`-mthumb` オプションは、`--target=aarch64-arm-none-eabi` などの AArch64 ターゲットでは無効です。AArch64 ターゲットでは、コンパイラは `-mthumb` オプションを無視し、警告を生成します。

デフォルト

ARMv8-A AArch32 および ARMv7-A ターゲットのデフォルトは `-marm` です。

例

```
armclang -c --target=arm-arm-none-eabi -march=armv8-a -mthumb test.c
```

関連参照

1.24 -marm（1-38 ページ）.
1.37 --target（1-52 ページ）.
1.26 -mcpu（1-40 ページ）.

関連情報

ターゲットアーキテクチャ、プロセッサ、および命令セットの指定.

1.31 -o

出力ファイルの名前を指定します。

オプション -o *filename* は、コンパイラが生成した出力ファイルの名前を指定します。

オプション -o- は、-c または -S オプションと共に使用すると、出力を標準出力ストリームに転送します。

デフォルト

-o オプションを指定しない場合は、コンパイラは、以下のテーブルの説明に従って、出力ファイルに名前を付けます。

表 1-1 -o オプションを使用しないコンパイル

コンパイラオプション	アクション	使用に関する注意事項
-c	ファイル拡張子 .o が付けられた入力ファイル名がデフォルトで使用されるオブジェクトファイルを生成します。	
-S	ファイル拡張子 .s が付けられた入力ファイル名がデフォルトで使用される出力ファイルを生成します。	
-E	プリプロセッサの出力を標準出力ストリームに書き込みます。	
（オプションなし）	一時オブジェクトファイルを生成し、リンカを自動的に呼び出して、デフォルト名が a.out の実行可能イメージを生成します。	コマンドラインでは、-o、-c、-E または -S は指定されません。

1.32 -O

ソースファイルのコンパイル時に使用する最適化レベルを指定します。

構文

`-O level`

level には、以下のいずれかを指定します。

0

最小限の最適化。ほとんどの最適化を実行しません。生成されるコードの構造がソースコードに直接対応しているため、デバッグ機能が有効になっている場合に、このオプションを選択すると、最良のデバッグビューが実現します。

これがデフォルトの最適化レベルです。

1

制限された最適化。デバッグ機能が有効になっている場合に、このオプションを選択すると、十分満足できるデバッグビューと優れたコード密度を実現できます。

2

高度な最適化。デバッグが有効の場合、オブジェクトコードとソースコードとの間のマップが明確でない場合があるため、デバッグビューの質は低下します。コンパイラは、デバッグ情報で説明できない最適化を実行する場合があります。

3

最大限の最適化。デバッグ機能が有効になっている場合に、このオプションを選択すると、質の低いデバッグビューになります。ARM では、最適化レベルを低くしてデバッグすることを推奨します。

fast

言語標準への厳密な準拠に違反する可能性のあるその他の強力な最適化と共に、-O3 の最適化をすべて有効にします。

s

コードサイズを縮小するために最適化を実行し、コード速度に対するコードサイズのバランスを取ります。

z

最適化を実行してイメージサイズを最小化します。

デフォルト

`-O`*level* を指定しない場合は、`-O0` が想定されます。

1.33 -rdynamic

実行可能ファイルにダイナミックシンボルが含まれている場合、シンボルを参照するのでなく、外から見えるすべてのシンボルをエクスポートします。

`armclang` はこのオプションを `--export-dynamic` に変換し、`armlink` に渡します。

`--export-dynamic` リンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

関連情報

ARM コンパイラツールチェーンリンカリファレンス.

1.34 -S

コンパイラによって生成されたマシンコードの逆アセンブリを出力します。

オブジェクトモジュールは生成されません。アセンブリ出力ファイルの名前は、デフォルトでは *filename*.s となり、現在の場所に配置されます。 *filename* はディレクトリ名を取り除いたソースファイルの名前です。デフォルトのファイル名は、-o オプションでオーバーライドできます。

関連参照

1.31 -o (1-46 ページ).

1.35 -std

コンパイルする言語標準を指定します。

構文

`-std=` *name*

各項目には以下の意味があります。

name

言語モードを指定します。有効値には次が含まれます。

c90

1990 C 標準によって定義される C。

gnu90

1990 C 標準に従った C で、GNU 拡張機能を追加しています。

c99

1999 C 標準によって定義される C。

gnu99

追加の GNU 拡張機能が付いた、1999 C 標準の定義に従った C です。

c11

2011 C 標準の定義に従った C です。

gnu11

追加の GNU 拡張機能が付いた、2011 C 標準の定義に従った C です。

c++98

1998 標準の定義に従った C++ です。

gnu++98

追加の GNU 拡張機能が付いた、1998 標準の定義に従った C++ です。

c++11

2011 標準の定義に従った C++ です。

gnu++11

追加の GNU 拡張機能が付いた、2011 標準の定義に従った C++ です。

C++ コードの場合、デフォルトは `gnu++98` です。C++ サポートの詳細については、Clang の Web サイトの *C++ Status* を参照して下さい。

C コードの場合、デフォルトは `gnu11` です。C サポートの詳細については、Clang の Web サイトの *言語互換性*を参照して下さい。

関連参照

1.45 -x(1-60 ページ).

関連情報

言語互換性.
C++ Status.

1.36 -stdlib

C++ ライブラリを指定して、システムインクルードパスに追加します。

構文

`-stdlib=`*`library_option`*

`library_option` は、以下のいずれかになります。

`libc++`
デフォルトのライブラリです。`include/libcxx` を C++ システムインクルードパスに追加します。

`legacy_cpplib`
`include/cpplib` を C++ システムインクルードパスに追加します。このオプションは、C++ Rogue Wave ライブラリ用です。

注

- 1 回の操作でコンパイルを実行しリンクする場合、`armclang` は `--stdlib=`*`library_option`* を `armlink` に自動的に渡します。
- clang オプション `-stdlib=libstdc++` はサポートされていません。

関連情報

--stdlib.

1.37 --target

特定の 3 重のターゲットのコードを生成します。

構文

`--target=` *`triple`*

各項目には以下の意味があります。

`triple`
`architecture-vendor-OS-abi` の形式が使用されます。

サポートされているターゲットは以下のとおりです。

`aarch64-arm-none-eabi`
AArch64 状態の A64 命令を生成します。`-mcpu` が指定されている場合を除き、`-march=armv8-a` を指定したと見なされます。

`arm-arm-none-eabi`
AArch32 状態の A32/T32 命令を生成します。`-march`(アーキテクチャをターゲットにする場合)または `-mcpu`(プロセッサをターゲットにする場合)と共に使用する必要があります。

注

- ターゲットでは大文字と小文字が区別されます。
- `--target` オプションは、`armclang` オプションです。その他すべてのツール(`armasm` や `armlink` など)では、`--cpu` および `--fpu` オプションを使用して、ターゲットプロセッサやターゲットアーキテクチャを指定して下さい。

デフォルト

`--target` オプションは必須であり、デフォルトはありません。必ずターゲットを 3 つ指定して下さい。

関連参照

1.24 -marm(1-38 ページ).
1.30 -mthumb(1-45 ページ).
1.26 -mcpu(1-40 ページ).
1.26 -mcpu(1-40 ページ).
1.28 -mfpu(1-43 ページ).

関連情報

ターゲットアーキテクチャ、プロセッサ、および命令セットの指定.
armasm ユーザガイド.
armlink ユーザガイド.

1.38 -u

未定義のシンボルがある場合に、指定されたシンボルを削除できないようにします。

構文

`-u` *`symbol`*

ここで、 *`symbol`* は保持するシンボルです。

`armclang` はこのオプションを `--undefined` に変換し、`armlink` に渡します。

`--undefined` リンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

関連情報

ARM コンパイラツールチェーンリンカリファレンス.

1.39 -v

コンパイラとリンカを呼び出すコマンドを表示し、それらのコマンドを実行します。

使用法

`-v` コンパイラオプションは、コンパイラとリンカを呼び出す具体的な方法を示す診断出力を生成し、各ツールのオプションを表示します。また、`-v` コンパイラオプションはバージョン情報も表示します。

`-v` オプションを使用すると、`armclang` はこの診断出力を表示し、コマンドを実行します。

注

コマンドを実行せずに診断出力を表示するには、`-###` オプションを使用します。

関連参照

1.46 -###(1-61 ページ).

1.40 --version

バージョン情報を表示します。

1.41 --version_number

使用している armclang のバージョンを表示します。

使用法

コンパイラは、nnnbbb 形式のバージョン番号を表示します。各項目には以下の意味があります。

- nnn はバージョン番号です。
- bbbb はビルド番号を示します。

関連参照

5.1 定義済みマクロ(5-114 ページ).

1.42 -W

診断を制御します。

構文

`-W name`

name の共通の値には以下のようなものがあります。

`-Werror`
: 警告をエラーに変えます。

`-Werror=foo`
: 警告 `foo` をエラーに代えます。

`-Wno-error=foo`
: `-Werror` が指定されている場合でも、警告 `foo` を警告のままにします。

`-Wfoo`
: 警告 `foo` を有効にします。

`-Wno-foo`
: 警告 `foo` を非表示にします。

`-Weverything`
: すべての警告を有効にします。

`armclang` を使用した診断の制御の詳細については、*「Clang Compiler User's Manual」*の *「Controlling Errors and Warnings」*を参照してください。

関連情報

armclang を使用して診断を制御するためのオプション.

1.43 -Wl

コンパイル後にリンク手順が実行されるときにリンカに渡すコマンドラインオプションを指定します。

使用可能なリンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

構文

`-Wl, opt,[opt[,...]]`

各項目には以下の意味があります。

opt

リンカに渡されるコマンドラインオプションです。

オプションまたは `option=argument` ペアのコンマ区切りリストを指定できます。

制約条件

`-Wl` がサポートされていないオプションを渡すと、リンカはエラーを生成します。

例

以下の例は、さまざまな構文の使用方法を示します。`armlink` は単一のオプション `--list=diag.txt` と 2 つのオプション `--list diag.txt` を同等に扱うため、それらは同等です。

```
armclang --target=aarch64-arm-none-eabi hello.c -Wl,--split,--list,diag.txt
armclang --target=aarch64-arm-none-eabi hello.c -Wl,--split,--list=diag.txt
```

関連参照

1.44 -Xlinker（1-59 ページ）.

1.44 -Xlinker

コンパイル後にリンク手順が実行されるときにリンカに渡すコマンドラインオプションを指定します。

使用可能なリンカオプションについては、『*ARM® コンパイラ ツールチェーンリンカリファレンス*』を参照してください。

構文

`-Xlinker` *opt*

各項目には以下の意味があります。

opt
リンカに渡されるコマンドラインオプションです。

複数のオプションを渡す場合は、複数の `-Xlinker` オプションを使用します。

制約条件

`-Xlinker` がサポートされていないオプションを渡すと、リンカはエラーを生成します。

例

この例では、オプション `--split` を以下のリンカに渡します。

```
armclang --target=aarch64-arm-none-eabi hello.c -Xlinker --split
```

この例では、オプション `--list diag.txt` を以下のリンカに渡します。

```
armclang --target=aarch64-arm-none-eabi hello.c -Xlinker --list -Xlinker diag.txt
```

関連参照

1.43 -Wl（1-58 ページ）.

1.45 -x

ソースファイルの言語を指定します。

構文

`-x` *Language*

各項目には以下の意味があります。

言語

次のソースファイルの言語を以下から 1 つ指定します。

`c`

C コード。

`c++`

C++ コード。

`assembler-with-cpp`

C プリプロセッサが必要な C ディレクティブを含むアセンブリコード。

アセンブラ

C プリプロセッサが不要なアセンブリコード。

使用法

`-x` は、コマンドライン上でその後に続く入力ファイルのデフォルトの言語標準をオーバーライドします。例えば、

```
armclang inputfile1.s -xc inputfile2.s inputfile3.s
```

この例では、`armclang` は入力ファイルを以下のように扱います。

- `inputfile1.s` は、`-xc` オプションの前に表示されます。`armclang` が、これをアセンブリコードとして扱うのは、接尾文字 `.s` があるためです。
- `inputfile2.s` と `inputfile3.s` は、`-xc` オプションの後に表示されるため、`armclang` は、C コードとして扱います。

注

`-std` を使用して、デフォルトの言語標準を設定します。

デフォルト

コンパイラは、デフォルトでは、以下のようにファイル名の接尾文字からソースファイルの言語を決定します。

- `.cpp`、`.cxx`、`.c++`、`.cc`、および `.CC` は、C++ を示します。`-x c++` と同等です。
- `.c` は C を示します。`-x c` と同等です。
- `.s`(小文字)は、プリプロセッシングの必要ないアセンブリコードを示します。`-x assembler` と同等です。
- `.S`(大文字)は、プリプロセッシングの必要なアセンブリコードを示します。`-x assembler-with-cpp` と同等です。

関連参照

1.35 -std(1-50 ページ).

1.46 -###

コンパイラとリンカを呼び出すコマンドを、それらのコマンドを実行せずに表示します。

使用法

`-###` コンパイラオプションは、コンパイラとリンカを呼び出す具体的な方法を示す診断出力を生成し、各ツールのオプションを表示します。また、`-###` コンパイラオプションはバージョン情報も表示します。

`-###` オプションを使用した場合、`armclang` はこの診断出力を表示するだけです。`armclang` はソースファイルをコンパイルしたり、`armlink` を呼び出したりしません。

注

診断出力を表示し、コマンドを実行するには、`-v` オプションを使用します。

関連参照

1.39 -v (1-54 ページ).

第 2 章
コンパイラ固有のキーワードおよび演算子

C および C++ 標準の拡張機能であるコンパイラ固有のキーワードおよび演算子の概要を示します。

以下のセクションから構成されています。

2.1 コンパイラ固有のキーワードおよび演算子

ARM コンパイラである armclang は、C および C++ 標準の拡張機能であるキーワードを提供します。

ARM コンパイラ固有の動作または制約条件をサポートしていない標準 C および標準 C++ キーワードは、記載されていません。

ARM コンパイラがサポートするキーワード拡張は、以下のとおりです。

- `__alignof__`
- `__asm`
- `__declspec`
- `__inline`

関連参照

2.2 __alignof__（2-64 ページ）.
2.3 __asm（2-66 ページ）.
2.4 __declspec 属性（2-67 ページ）.
2.8 __inline（2-71 ページ）.

2.2 __alignof__

__alignof__ キーワードを使用すると、型や変数の境界整列について照会できます。

—————— 注 ——————
このキーワードは、ARM コンパイラでサポートされている GNU コンパイラ拡張機能です。
————————————

構文

__alignof__(*type*)

__alignof__(*expr*)

各項目には以下の意味があります。

タイプ
　型を指定します。
expr
　左辺値を指定します。

戻り値

__alignof__(*type*) は type 型の整列要件を返し、整列要件がない場合は 1 を返します。

__alignof__(*expr*) は、左辺値 *expr* の型の境界整列要件を返し、境界整列要件がない場合は 1 を返します。

例

以下の例では、まずデータ型から直接、次に対応するデータ型の左辺値からと、さまざまなデータ型の整列要求を示します。

```
#include <stdio.h>

int main(void)
{
  int       var_i;
  char      var_c;
  double    var_d;
  float     var_f;
  long      var_l;
  long long var_ll;

  printf("Alignment requirement from data type:\n");
  printf("  int       :%d\n", __alignof__(int));
  printf("  char      :%d\n", __alignof__(char));
  printf("  double    :%d\n", __alignof__(double));
  printf("  float     :%d\n", __alignof__(float));
  printf("  long      :%d\n", __alignof__(long));
  printf("  long long :%d\n", __alignof__(long long));
  printf("\n");
  printf("Alignment requirement from data type of lvalue:\n");
  printf("  int       :%d\n", __alignof__(var_i));
  printf("  char      :%d\n", __alignof__(var_c));
  printf("  double    :%d\n", __alignof__(var_d));
  printf("  float     :%d\n", __alignof__(var_f));
  printf("  long      :%d\n", __alignof__(var_l));
  printf("  long long :%d\n", __alignof__(var_ll));
}
```

以下のコマンドでコンパイルすると、以下の出力が生成されます。

```
armclang --target=arm-arm-none-eabi -march=armv8-a alignof_test.c -o alignof.axf
```

```
データ型の整列要求は以下のとおりです。
  int       :4
  char      :1
  double    :8
  float     :4
  long      :4
```

```
  long long :8
```

左辺値のデータ型の整列要求は以下のとおりです。

```
  int       :4
  char      :1
  double    :8
  float     :4
  long      :4
  long long :8
```

2.3 __asm

このキーワードは、情報を armclang アセンブラに渡します。

このキーワードの正確なアクションは、使用法によって異なります。

使用法

インラインアセンブリ

__asm キーワードを使用して、インライン GCC 構文アセンブリコードを関数に組み込むことができます。以下に例を示します。

```
#include <stdio.h>

int add(int i, int j)
{
  int res = 0;
  __asm (
    "ADD %[result], %[input_i], %[input_j]"
    :[result] "=r" (res)
    :[input_i] "r" (i), [input_j] "r" (j)
  );
  return res;
}

int main(void)
{
  int a = 1;
  int b = 2;
  int c = 0;

  c = add(a,b);

  printf("Result of %d + %d = %d\n", a, b, c);
}
```

__asm インラインアセンブリステートメントの一般的な形式は以下のとおりです。

__asm(*code* [:*output_operand_list* [:*input_operand_list* [:*clobbered_register_list*]]]);

code はアセンブリコードです。例では、これは "ADD %[result], %[input_i], %[input_j]" です。

output_operand_list は、コンマで区切られた、出力オペランドのオプションのリストです。各オペランドは、角括弧で囲まれたシンボリック名、制約文字列、括弧で囲まれた C 式で構成されます。例として、[result] "=r" (res) という 1 つの出力オペランドがあります。

input_operand_list は、コンマで区切られたオプションの入力オペランドのリストです。入力オペランドは、出力オペランドと同じ構文を使用します。例では、2 つの入力オペランドがあります。[input_i] "r" (i)、[input_j] "r" (j)。

clobbered_register_list は、上書きされたレジスタのオプションリストです。例では、これは省略します。

アセンブリラベル

__asm キーワードを使用して、C シンボルのアセンブリラベルを指定できます。以下に例を示します。

```
int count __asm__("count_v1"); // count ではなく count_v1 をエクスポート
```

2.4 __declspec 属性

__declspec キーワードを使用すると、オブジェクトおよび関数の特殊な属性を指定できます。

__declspec キーワードは、宣言の指定の先頭に付ける必要があります。以下に例を示します。

```
__declspec(noreturn) void overflow(void);
```

使用できる __declspec 属性は以下のとおりです。

- __declspec(noinline)
- __declspec(noreturn)
- __declspec(nothrow)

__declspec 属性は、記憶域クラス修飾子です。これらは、関数または変数の型には影響しません。

関連参照

2.5 __declspec(noinline) (2-68 ページ).
2.6 __declspec(noreturn) (2-69 ページ).
2.7 __declspec(nothrow) (2-70 ページ).

2.5 __declspec(noinline)

__declspec(noinline) を使用すると、その関数の呼び出し時に、その関数をインライン展開しないようにすることができます。

__declspec(noinline) を定数データに適用して、コンパイラによる最適化を抑制することもできます。オブジェクト内の値の配置には影響しません。この機能は、パッチ可能な定数（後で別の値に適用されるデータ）に使用することができます。定数値が必須であるようなコンテキストでこのような定数を使用するのは誤った用法です（配列の次元など）。

注

この __declspec 属性には、同等の関数属性 __attribute__((noinline)) があります。

例

```
/* y の最適化を抑制 */
__declspec(noinline) const int y = 5;
/* foo() のインライン展開を抑制 */
__declspec(noinline) int foo(void);
```

2.6 __declspec(noreturn)

__declspec(noreturn) 属性は、関数が決して制御を返さないことを示します。

——— 注 ———

この __declspec 属性には、同等の関数属性 __attribute__((noreturn)) があります。

———————————

使用法

この属性を使用して、exit() など、戻らない関数を呼び出すコストを削減します。noreturn 関数がその呼び出し元に戻る場合の動作は定義されていません。

制約条件

noreturn 関数の呼び出し時に、復帰アドレスは保持されません。これにより、デバッガによるコールスタックの表示が制限されます。

例

```
__declspec(noreturn) void overflow(void); // overflow は決して制御を返さない
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

2.7 __declspec(nothrow)

__declspec(nothrow) 属性は、関数の呼び出しによって C++例外が被発呼側から呼び出し元に伝播されないことを示します。

ARM ライブラリヘッダにより、この修飾子は、ISO C 標準で例外をスローしないことが規定されている C 関数の宣言に自動的に追加されます。ただし、C++ コンテキストに例外をスローする可能性がある C ライブラリ関数に対して生成される unwind テーブルにいくつかの制限があります。例えば、bsearch や qsort です。

注

この __declspec 属性には、同等の関数属性 __attribute__((nothrow)) があります。

使用法

コンパイラは、関数によって例外がスローされないことを認識している場合、その関数の呼び出し元用に生成する例外処理テーブルのサイズを縮小できることがあります。

制限

関数の呼び出しによって C++例外が被発呼側から呼び出し元に伝播される場合、動作は定義されません。

例外がイネーブルになった状態でコンパイルしないと、この修飾子は無視されます。

例

```
struct S {     ~S(); }; __declspec(nothrow) extern void f(void); void g(void) {     S s;
f(); }
```

関連情報

標準 C++ ライブラリの実装定義.

2.8 __inline

__inline キーワードは、コンパイラに対し、必要に応じて、C 関数または C++ 関数をインラインでコンパイルするように指示します。

__inline は C90 コードで使用でき、C99 inline キーワードの代わりの役割を果たします。

__inline と __inline__ の両方は、armclang でサポートされています。

例

```
static __inline int f(int x){
    return x*5+1;
}

int g(int x, int y){
    return f(x) + f(y);
}
```

関連概念

5.2 インライン関数(5-117 ページ).

第 3 章
コンパイラ固有の関数、変数、および型属性

C および C++ 標準の拡張機能であるコンパイラ固有の関数、変数、および型属性の概要を示します。

以下のセクションから構成されています。

3.1 関数属性

__attribute__ キーワードを使用すると、変数、構造体フィールド、関数、型などの特殊な属性を指定できます。

このキーワード形式には、以下のいずれかを使用します。

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

以下に例を示します。

```
int my_function(int b) __attribute__((const));
static int my_variable __attribute__((__unused__));
```

以下のテーブルは、使用できる関数属性の概要が記載されています。

表 3-1 コンパイラがサポートする関数属性および同等の属性

関数属性	この属性以外の同等キーワード
__attribute__((alias))	-
__attribute__((always_inline))	-
__attribute__((const))	-
__attribute__((constructor[(*priority*)]))	-
__attribute__((deprecated))	-
__attribute__((destructor[(*priority*)]))	-
__attribute__((format_arg(*string-index*)))	-
__attribute__((malloc))	-
__attribute__((noinline))	__declspec(noinline)
__attribute__((nomerge))	-
__attribute__((nonnull))	-
__attribute__((noreturn))	__declspec(noreturn))
__attribute__((nothrow))	__delspec(nothrow)
__attribute__((notailcall))	-
__attribute__((pcs("*calling_convention*")))	-
__attribute__((pure))	-
__attribute__((section("*name*")))	-
__attribute__((unused))	-
__attribute__((used))	-
__attribute__((visibility("*visibility_type*")))	-
__attribute__((weak))	-
__attribute__((weakref("target")))	-

使用法

宣言、定義、または両方でこれらの関数属性を指定できます。以下に例を示します。

```
void AddGlobals(void) __attribute__((always_inline));
__attribute__((always_inline)) void AddGlobals(void) {...}
```

関数属性が競合する場合、コンパイラは安全で強い属性を使用します。たとえば、`__attribute__((used))` は `__attribute__((unused))` より安全で、`__attribute__((noinline))` は `__attribute__((always_inline))` より安全です。

関連参照

3.2 __attribute__((always_inline)) 関数属性(3-76 ページ).
3.3 __attribute__((const)) 関数属性(3-77 ページ).
3.4 __attribute__((constructor[(priority)])) 関数属性(3-78 ページ).
3.5 __attribute__((format_arg(string-index))) 関数属性(3-79 ページ).
3.6 __attribute__((malloc)) 関数属性(3-80 ページ).
3.8 __attribute__((nonnull)) 関数属性(3-82 ページ).
3.7 __attribute__((noinline)) 関数属性(3-81 ページ).
3.11 __attribute__((pcs("calling_convention"))) 関数属性(3-85 ページ).
3.12 __attribute__((pure)) 関数属性(3-86 ページ).
3.9 __attribute__((noreturn)) 関数属性(3-83 ページ).
3.10 __attribute__((nothrow)) 関数属性(3-84 ページ).
3.13 __attribute__((section("name"))) 関数属性(3-87 ページ).
3.15 __attribute__((unused)) 関数属性(3-89 ページ).
3.14 __attribute__((used)) 関数属性(3-88 ページ).
3.16 __attribute__((visibility("visibility_type"))) 関数属性(3-90 ページ).
3.17 __attribute__((weak)) 関数属性(3-91 ページ).
3.18 __attribute__((weakref("target"))) 関数属性(3-92 ページ).
2.2 __alignof__(2-64 ページ).
2.3 __asm(2-66 ページ).
2.4 __declspec 属性(2-67 ページ).

3.2 __attribute__((always_inline)) 関数属性

この関数属性は、関数をインライン展開する必要があることを示します。

コンパイラは、関数の特性に関係なく、その関数をインライン関数にしようとします。ただし、問題が発生する場合、コンパイラは関数をインライン関数にしません。

例

```
static int max(int x, int y) __attribute__((always_inline));
static int max(int x, int y)
{
    return x > y ? x :y; // 可能な場合は常にインライン展開
}
```

3.3 __attribute__((const)) 関数属性

const 関数属性は、関数が引数のみを確認し、戻り値以外には作用しないことを指定します。つまり、関数はグローバルメモリの読み出しまたは変更を行いません。

渡された引数に対してのみ演算を行う関数は、共通部分式の削除とループの最適化の対象になります。

関数によるグローバルメモリの読み出しは許可されていないため、これは __attribute__((pure)) よりも厳密なクラスです。

例

```
#include <stdio.h>

// __attribute__((const)) functions do not read or modify any global memory
int my_double(int b) __attribute__((const));
int my_double(int b) {
  return b*2;
}

int main(void) {
    int i;
    int result;
    for (i = 0; i < 10; i++)
    {
       result = my_double(i);
       printf (" i = %d ; result = %d \n", i, result);
    }
}
```

3.4 __attribute__((constructor[(priority)])) 関数属性

この属性を指定すると、関連付けられている関数が、main() に入る前に自動的に呼び出されます。

構文

```
__attribute__((constructor[( priority )]))
```

priority は、優先度を示す整数値（省略可）です。割り当てられている整数値が小さいコンストラクタから先に実行されます。優先度が割り当てられていないコンストラクタは、優先度が割り当てられているコンストラクタの後に実行されます。

100 以下の優先度値は内部使用専用に予約されています。これらの値を使用すると、コンパイラから警告が生成されます。

使用法

この属性は、起動コードまたは初期化コードに対して使用できます。例えば、DLL のロード時に呼び出す関数に使用します。

例

以下の例では、main() が実行される前に、指定された順序でコンストラクタ関数が呼び出されます。

```
void my_constructor1(void) __attribute__((constructor));
void my_constructor2(void) __attribute__((constructor(102)));
void my_constructor3(void) __attribute__((constructor(103)));
void my_constructor1(void) /* This is the 3rd constructor */
{                         /* 呼び出される関数 */
    printf("Called my_constructor1()\n");
}
void my_constructor2(void) /* これは最初のコンストラクタです */
{                          /* 呼び出される関数 */
    printf("Called my_constructor2()\n");
}
void my_constructor3(void) /* これは 2 番目のコンストラクタです */
{                          /* 呼び出される関数 */
    printf("Called my_constructor3()\n");
}
int main(void)
{
    printf("Called main()\n");
}
```

このサンプルは、以下の出力を生成します。

```
Called my_constructor2()
Called my_constructor3()
Called my_constructor1()
Called main()
```

3.5 __attribute__((format_arg(string-index))) 関数属性

この属性では、関数が引数としてフォーマットストリングを取るように指定されます。フォーマットストリングには、printf()、scanf()、strftime()、strfmon() などの printf-style 関数に渡されることを意図した、型指定されたプレースホルダを含めることができます。

この属性により、コンパイラは、この関数の出力が printf-style 関数の呼び出しに使用される場合に、指定された引数でプレースホルダの型の確認を行います。

構文

```
__attribute__((format_arg( string-index )))
```

ここで、 *string-index* は、フォーマットストリング引数（1 から開始）である引数を指定します。

例

以下のサンプルでは、フォーマットストリングを printf() に提供する 2 つの関数 myFormatText1() と myFormatText2() を宣言します。

1 つ目の関数 myFormatText1() は format_arg 属性を指定しません。コンパイラは、フォーマットストリングとの一貫性について printf 引数の型を確認しません。

2 つ目の関数 myFormatText2() は format_arg 属性を指定します。printf() へのその後の呼び出しで、コンパイラは、指定された引数 a および b の型と myFormatText2() へのフォーマットストリング引数との一貫性を確認します。int が期待されるときに float が指定されると、コンパイラは警告を表示します。

```
#include <stdio.h>

// printf が使用する関数。フォーマットの型の確認なし。
extern char *myFormatText1 (const char *);

// printf が使用する関数。引数 1 でフォーマットの型を確認。
extern char *myFormatText2 (const char *) __attribute__((format_arg(1)));


int main(void) {
  int a;
  float b;

  a = 5;
  b = 9.099999;

  printf(myFormatText1("Here is an integer: %d\n"), a); // 型の確認なし。型は一致します。
  printf(myFormatText1("Here is an integer: %d\n"), b); // 型の確認なし。型は一致しませんが、警告なし

  printf(myFormatText2("Here is an integer: %d\n"), a); // 型を確認。型は一致しています。
  printf(myFormatText2("Here is an integer: %d\n"), b); // 型を確認。型の不一致のため警告が生成されます。
}


$ armclang --target=aarch64-arm-none-eabi -c format_arg_test.c
format_arg_test.c:21:53: warning: format specifies type 'int' but the argument has type
'float' [-Wformat]
  printf(myFormatText2("Here is an integer: %d\n"), b); // 型を確認。型の不一致のため警告が生成されます。
                                                ~~      ^
                                                %f
1 warning generated.
```

3.6 __attribute__((malloc)) 関数属性

malloc 関数と同じように処理し、関連付けられた最適化をコンパイラが実行できることを示す関数属性です。

例

```
void * foo(int b) __attribute__((malloc));
```

3.7 __attribute__((noinline)) 関数属性

この属性を使用すると、その関数の呼び出し時に、その関数をインライン展開しないようにすることができます。

`__attribute__((noinline))` を定数データに適用して、コンパイラによる最適化を抑制することもできます。オブジェクト内の値の配置には影響しません。この機能は、パッチ可能な定数（後で別の値に適用されるデータ）に使用することができます。定数値が必須であるようなコンテキストでこのような定数を使用するのは誤った用法です

例

```
/* y の最適化を抑制 */
const int y = 5 __attribute__((noinline));
/* foo() のインライン展開を抑制 */
int foo(void) __attribute__((noinline));
```

3.8 __attribute__((nonnull)) 関数属性

この関数属性は、NULL ポインタをサポートしない関数パラメータを指定します。該当するパラメータが検出されたときにコンパイラから警告を生成することができます。

構文

`__attribute__((nonnull[(arg-index, ...)]))`

`[(arg-index, ...)]` は、オプションの引数インデックスリストを示します。

引数インデックスリストが指定されなかった場合、すべてのポインタ引数が nonnull としてマークされます。

例

以下の宣言は等価です。

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull (1, 2)));
```

```
void * my_memcpy (void *dest, const void *src, size_t len) __attribute__((nonnull));
```

3.9 __attribute__((noreturn)) 関数属性

この属性は、関数が戻らないことを示します。

使用法

この属性を使用して、exit() など、戻らない関数を呼び出すコストを削減します。noreturn 関数がその呼び出し元に戻る場合の動作は定義されていません。

制約条件

noreturn 関数の呼び出し時に、復帰アドレスは保持されません。これにより、デバッガによるコールスタックの表示が制限されます。

3.10 __attribute__((nothrow)) 関数属性

この属性は、関数の呼び出しによって C++ 例外が呼び出し先から呼び出し元に送信されないことを示します。

ARM ライブラリヘッダにより、この修飾子は、ISO C 標準で例外をスローしないことが規定されている C 関数の宣言に自動的に追加されます。ただし、C++ コンテキストに例外をスローする可能性がある C ライブラリ関数に対して生成される unwind テーブルにいくつかの制限があります。例えば、`bsearch` や `qsort` です。

コンパイラは、関数によって例外がスローされないことを認識している場合、その関数の呼び出し元用に生成する例外処理テーブルのサイズを縮小できることがあります。

3.11 __attribute__((pcs("calling_convention"))) 関数属性

この関数属性は、ハードウェア浮動小数点を持つターゲットの呼び出し規則を指定します。

構文

```
__attribute__((pcs(" calling_convention ")))
```

calling_convention は、以下のいずれかになります。

aapcs
: 整数レジスタを使用します。

aapcs-vfp
: 浮動小数点レジスタを使用します。

例

```
double foo (float) __attribute__((pcs("aapcs")));
```

3.12 __attribute__((pure)) 関数属性

多くの関数は、値を返す以外に効果がなく、その戻り値はパラメータとグローバル変数にのみ依存します。このような関数は、データフロー分析の対象となり、削除される場合があります。

例

```
int bar(int b) __attribute__((pure));
int bar(int b)
{
    return b++;
}
int foo(int b)
{
    int aLocal=0;
    aLocal += bar(b);
    aLocal += bar(b);
    return 0;
}
```

この場合、bar の結果は使用されないため、この関数の呼び出しは削除することができます。

3.13 __attribute__((section("name"))) 関数属性

section 関数属性を使用すると、イメージの異なるセクションにコードを配置できます。

例

以下の例では、関数 foo は、.text ではなく、new_section という RO セクションに置かれます。

```
int foo(void) __attribute__((section ("new_section")));
int foo(void)
{
  return 2;
}
```

3.14 __attribute__((used)) 関数属性

この関数属性を使用すると、そのスタティック関数は参照されていなくても、オブジェクトファイル内に保持されることをコンパイラに通知できます。

`__attribute__((used))` でマークされた関数には、オブジェクトファイル内で、リンカによる未使用のセクションの削除処理を抑制するためのタグが付けられます。

注

`__attribute__((used))` を使用して、スタティック変数を使用済みとマークすることもできます。

例

```
static int lose_this(int); static int keep_this(int) __attribute__((used));     // オブジェクトファイル内に保持される static int keep_this_too(int) __attribute__((used)); // オブジェクトファイル内に保持される
```

3.15 __attribute__((unused)) 関数属性

unused 関数属性を使用すると、その関数が参照されない場合に警告が生成されないようにすることができます。この関数属性を使用しても、未使用の関数を削除する処理の動作が変更されることはありません。

注

デフォルトで、コンパイラは未使用の関数について警告しません。-Wunused-function を使用して、この警告を特に有効にするか、または、-Wall などの装備された -W 値を使用します。

__attribute__((unused)) 属性は、未使用の関数について警告する場合に役に立つ場合がありますが、特定の関数セットについての警告を非表示にします。

例

```
static int unused_no_warning(int b) __attribute__((unused));
static int unused_no_warning(int b)
{
  return b++;
}

static int unused_with_warning(int b);
static int unused_with_warning(int b)
{
  return b++;
}
```

-Wall を使用してこの例をコンパイルすると、以下の警告が生成されます。

```
armclang --target=aarch64-arm-none-eabi -c test.c -Wall
test2.cpp:10:12: warning: unused function 'unused_with_warning' [-Wunused-function]
static int unused_with_warning(int b)
           ^
1 warning generated.
```

関連参照

3.30 __attribute__((unused)) 変数属性(3-104 ページ).

3.16 __attribute__((visibility("visibility_type"))) 関数属性

この関数属性は、ELF のシンボルの可視性に作用します。

構文

```
__attribute__((visibility(" visibility_type ")))
```

visibility_type は、以下のいずれかになります。

デフォルト
: シンボルに想定される可視性は他のオプションによって変更される場合があります。default の可視性は、このような変更よりも優先されます。default の可視性は、外部リンケージに対応します。

`hidden`
: シンボルはダイナミックシンボルテーブルには配置されません。そのため、他の実行可能ファイルや共有ライブラリが直接参照することはできません。関数ポインタを使用した間接参照は可能です。

`protected`
: シンボルはダイナミックシンボルテーブルに配置されますが、それを定義しているモジュール内の参照は、ローカルシンボルにバインドされます。つまり、このシンボルが別のモジュールによってオーバーライドされることはありません。

使用法

この属性は、`default` の可視性を指定する場合を除き、暗黙的に外部リンケージが割り当てられるような宣言での使用が想定されています。

この属性を C および C++ の関数と変数に適用できます。C++ では、クラス、構造体、共用体、列挙型、およびネームスペース宣言に適用することもできます。

ネームスペース名の宣言の場合、可視性属性はすべての関数と変数定義に適用されます。

例

```
void __attribute__((visibility("protected"))) foo()
{
   ...
}
```

3.17 __attribute__((weak)) 関数属性

__attribute__((weak)) で定義された関数は、そのシンボルを weak でエクスポートします。

__attribute__((weak)) で宣言した後に __attribute__((weak)) を使用せずに定義した関数は、*weak* 関数として動作します。

例

```
extern int Function_Attributes_weak_0 (int b) __attribute__ ((weak));
```

3.18 __attribute__((weakref("target"))) 関数属性

この関数属性は、(それ自身にはターゲットシンボルの関数定義を必要としない)エイリアスとして関数宣言をマークします。

構文

```
__attribute__((weakref("target")))
```

target にはターゲットシンボルを指定します。

例

以下の例では、foo() が、弱参照を通じて y() を呼び出します。

```
extern void y(void); static void x(void) __attribute__((weakref("y"))); void foo (void)
{   ...   x();   ...}
```

制約条件

この属性は、静的リンケージがある関数でのみ使用できます。

3.19 型属性

__attribute__ キーワードを使用すると、変数または構造体フィールド、関数、型などの特殊な属性を指定できます。

このキーワード形式には、以下のいずれかを使用します。

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

以下に例を示します。

```
typedef union { int i; float f; } U __attribute__((transparent_union));
```

使用できる型属性は以下のとおりです。

- __attribute__((aligned))
- __attribute__((packed))
- __attribute__((transparent_union))

関連参照

3.20 __attribute__((aligned)) 型属性(3-94 ページ).
3.22 __attribute__((transparent_union)) 型属性(3-96 ページ).
3.21 __attribute__((packed)) 型属性(3-95 ページ).

3.20 __attribute__((aligned)) 型属性

aligned 型属性を使用すると、その型の最小アラインメントを指定できます。

3.21 __attribute__((packed)) 型属性

packed 型属性を使用すると、その型に可能な最小境界整列を使用するように指定できます。

3.22 __attribute__((transparent_union)) 型属性

transparent_union 型属性を使用すると、*transparent_union 型*を指定できます。

透過ユニオン型を持つパラメータを使用して関数が定義されると、共用体の型の引数を使用した関数の呼び出しは、渡された引数の型を持つメンバおよび渡された引数値に設定された値を持つ共用体オブジェクトを初期化する結果になります。

共用体データ型が __attribute__((transparent_union)) で修飾されると、透過ユニオンはその型のすべての関数パラメータに適用されます。

注

個別関数パラメータは、対応する __attribute__((transparent_union)) 変数属性で修飾することもできます。

例

```
typedef union { int i; float f; } U __attribute__((transparent_union));
void foo(U u)
{
    static int s;
    s += u.i;    /* 'int' フィールドを使用 */
}
void caller(void)
{
    foo(1);        /* u.i は 1 に設定される */
    foo(1.0f);     /* u.f は 1.0f に設定される */
}
```

3.23 変数属性

__attribute__ キーワードを使用すると、変数または構造体フィールド、関数、型などの特殊な属性を指定できます。

このキーワード形式には、以下のいずれかを使用します。

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__((__attribute1__, __attribute2__, ...))
```

以下に例を示します。

```
static int b __attribute__((__unused__));
```

使用できる変数属性は以下のとおりです。

- __attribute__((alias))
- __attribute__((aligned))
- __attribute__((deprecated))
- __attribute__((packed))
- __attribute__((section("name")))
- __attribute__((unused))
- __attribute__((used))
- __attribute__((weak))
- __attribute__((weakref("target")))

関連参照

3.24 __attribute__((alias)) 変数属性(3-98 ページ).
3.25 __attribute__((aligned)) 変数属性(3-99 ページ).
3.26 __attribute__((deprecated)) 変数属性(3-100 ページ).
3.27 __attribute__((packed)) 変数属性(3-101 ページ).
3.28 __attribute__((section("name"))) 変数属性(3-102 ページ).
3.30 __attribute__((unused)) 変数属性(3-104 ページ).
3.29 __attribute__((used)) 変数属性(3-103 ページ).
3.31 __attribute__((weak)) 変数属性(3-105 ページ).
3.32 __attribute__((weakref("target"))) 変数属性(3-106 ページ).

3.24 __attribute__((alias)) 変数属性

この変数属性を使用すると、変数の複数のエイリアスを指定できます。

エイリアスは、元の変数と同じ変換単位で定義しなければなりません。

——— 注 ———

エイリアスはブロックの有効範囲に指定できません。コンパイラは、ローカル変数の定義に付属しているエイリアスの属性を無視し、変数の定義を通常のローカル定義として扱います。

———————

出力オブジェクトファイルでは、コンパイラはエイリアスのリファレンスを元の変数名への参照に置き換え、エイリアスを元の名前と共に生成します。以下に例を示します。

```
int oldname = 1;
extern int newname __attribute__((alias("oldname")));
```

このコードは、以下にコンパイルされます。

```
        movw    r0, :lower16:newname
        movt    r0, :upper16:newname
        ldr     r1, [r0]
        ...
        .type   oldname,%object         @ @oldname
        .data
        .globl  oldname
        .align  2
oldname:
        .long   1                       @ 0x1
        .size   oldname, 4
        ...
        .globl  newname
newname = oldname
```

——— 注 ———

関数名は、対応する関数属性 `__attribute__((alias))` を使用してエイリアスすることができます。

———————

構文

`type` *newname* `__attribute__((alias("` *oldname* `")));`

各項目には以下の意味があります。

oldname
エイリアスされる変数の名前を指定します。

newname
エイリアスされる変数の新しい名前を指定します。

例

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // 宣言
void foo(void)
{
    printf("newname = %d\n", newname); // 1 を出力
}
```

3.25 __attribute__((aligned)) 変数属性

aligned 変数属性を使用すると、変数や構造体フィールドの最小境界整列をバイト単位で指定できます。

例

```
/* 16 バイト境界で整列 */
int x __attribute__ ((aligned (16)));

/* この場合、スカラデータ型の最大の境界整列が使用されます。ARM の場合は 8 バイトになります。*/
short my_array[3] __attribute__ ((aligned));
```

3.26 __attribute__((deprecated)) 変数属性

deprecated 変数属性を使用すると、警告やエラーが生成されることなく、非推奨の変数を宣言できます。deprecated 変数へのアクセスに対しては警告が生成されますが、コンパイルは実行されます。

この警告では、その変数が使用されている場所と定義されている場所を通知します。この情報は、特定の変数が非推奨の原因を判断するのに役立ちます。

例

```
extern int deprecated_var __attribute__((deprecated));
void foo()
{
    deprecated_var=1;
}
```

この例をコンパイルすると、以下の警告が生成されます。

```
armclang --target=aarch64-arm-none-eabi -c test_deprecated.c
test_deprecated.c:4:3: warning: 'deprecated_var' is deprecated [-Wdeprecated-declarations]
  deprecated_var=1;
  ^
test_deprecated.c:1:12: note: 'deprecated_var' declared here
  extern int deprecated_var __attribute__((deprecated));
  ^
1 warning generated.
```

3.27 __attribute__((packed)) 変数属性

packed 変数属性を使用すると、その変数や構造体フィールドに可能な最小アラインメントを使用するように指定できます。つまり、より大きい値を aligned 属性で指定しない限り、変数には 1 バイト、フィールドには 1 ビットが使用されます。

```
struct {     char a;     int b __attribute__((packed)); } Variable_Attributes_packed_0;
```

関連参照

3.25 __attribute__((aligned)) 変数属性(3-99 ページ).

3.28 __attribute__((section("*name*"))) 変数属性

section 属性を使用すると、変数を特定のデータセクションに配置するように指定できます。

通常、ARM コンパイラでは、生成したデータを .data や .bss などのセクションに配置します。しかし、追加のデータセクションが必要になったり、変数を特別なセクションに配置したりする場合（特別なハードウェアへのマップを行う場合など）があります。

section 属性を使用すると、読み出し専用変数は RO データセクションに配置され、書き込み可能変数は RW データセクションに配置されます。

セクション名が、.bss. で始まる場合、変数は ZI セクションに配置されます。

例

```
/* RO セクション */
const int descriptor[3] __attribute__ ((section ("descr"))) = { 1,2,3 };
/* RW セクション */
long long rw_initialized[10] __attribute__ ((section ("INITIALIZED_RW"))) = {5};
/* RW セクション */
long long rw[10] __attribute__ ((section ("RW")));
/* ZI セクション */
int my_zi __attribute__((section (".bss.my_zi_section")));
```

3.29 __attribute__((used)) 変数属性

この変数属性を使用すると、そのスタティック変数は参照されていなくても、オブジェクトファイル内に保持されることをコンパイラに通知できます。

`__attribute__((used))` でマークされたデータには、オブジェクトファイル内で、リンカによる未使用のセクションの削除処理を抑制するためのタグが付けられます。

注

`__attribute__((used))` を使用して、スタティック関数を使用済みとマークすることもできます。

例

```
static int lose_this = 1;
static int keep_this __attribute__((used)) = 2;     // オブジェクトファイルに保持されます
static int keep_this_too __attribute__((used)) = 3; // オブジェクトファイル内に保持される
```

3.30 __attribute__((unused)) 変数属性

宣言されている変数が一度も参照されない場合は、警告が生成される場合があります。
__attribute__((unused)) 属性を使用すると、特定の変数が使用されないことが予想済みであることをコンパイラに通知し、警告を生成しないように指定できます。

——— 注 ———

デフォルトでは、コンパイラは未使用の変数について警告しません。`-Wunused-variable` を使用して、この警告を特に有効にするか、または、`-Weverything` などの装備された `-W` 値を使用します。

`__attribute__((unused))` 属性は、未使用の変数について警告する場合に役に立つ場合がありますが、特定の変数セットについての警告を非表示にします。

例

```
void foo()
{
    static int aStatic =0;
    int aUnused __attribute__((unused));
    int bUnused;
    aStatic++;
}
```

適切な `-W` 設定でコンパイルされると、コンパイラは、`bUnused` は宣言されても一度も参照されないことを示す以下のような警告を生成しますが、`aUnused` に対しては警告を生成しません。

```
armclang --target=aarch64-arm-none-eabi -c test_unused.c -Wall
test_unused.c:5:7: warning: unused variable 'bUnused' [-Wunused-variable]
  int bUnused;
      ^
1 warning generated.
```

関連参照

3.15 __attribute__((unused)) 関数属性(3-89 ページ).

3.31 __attribute__((weak)) 変数属性

デフォルトの strong シンボルでなく、変数の weak シンボルを生成します。

```
extern int foo __attribute__((weak));
```

リンク時に、strong シンボルは weak シンボルをオーバーライドします。これにより、リンクに対してオブジェクトファイルの特定の組み合わせを選択することにより、weak シンボルを strong シンボルに置き換えることができます。

3.32 __attribute__((weakref("*target*"))) 変数属性

この変数属性は、(それ自身にはターゲットシンボルの定義を必要としない)エイリアスとして変数宣言をマークします。

構文

```
__attribute__((weakref("target")))
```

target にはターゲットシンボルを指定します。

例

以下の例では、弱参照を通じて、a に y の値を代入しています。

```
extern int y;
static int x __attribute__((weakref("y")));
void foo (void)
{
  int a = x;
  ...
}
```

制約条件

この属性は、static として宣言された変数に対してのみ使用できます。

第 4 章
コンパイラ固有のプラグマ

C および C++ 標準の拡張機能である ARM コンパイラ固有のプラグマを要約します。

以下のセクションから構成されています。

4.1 #pragma clang system_header

現在のファイル内の後続の宣言が、システムヘッダファイルにあるものとしてマークされます。

このプラグマを使用して、ファイルから生成された警告メッセージを宣言された時点から非表示にできます。

4.2 #pragma once

このプラグマを使用すると、コンパイラは後続のヘッダファイルのインクルードをスキップできます。

`#pragma once` は、他のコンパイラとの互換性を保つ目的でサポートされているため、他の形式のヘッダ保護コーディングを使用できます。しかし、`#ifndef` や `#define` のコーディングの方が移植が容易であるため、これらのディレクティブを使用することを推奨します。

例

以下のサンプルのように、ヘッダファイルの本体を囲むように `#ifndef` ガード変数を配置し、`#define` ガード変数を `#ifndef` の後に配置します。

```
#ifndef FILE_H #define FILE_H #pragma once          // オプション
 ... body of the header file ...#endif
```

上記の例では、`#pragma once` がオプションとしてマークされています。これは、コンパイラが `#ifndef` ヘッダ保護コーディングを認識して、`#pragma once` がなくてもそれ以後のインクルードをスキップするためです。

4.3 #pragma pack(*n*)

このプラグマを使用すると、構造体のメンバの境界が *n* と自然境界整列の小さい方の値で整列されます。パックされたオブジェクトの読み出しと書き込みは、非境界整列アクセスを使用して行われます。

——— 注 ———

このプラグマは、ARM コンパイラがサポートする GNU コンパイラ拡張機能です。

———————————

構文

```
#pragma pack( n )
```

各項目には以下の意味があります。

n

バイト単位の境界整列です。有効な境界整列値は 1、2、4、および 8 です。

デフォルト

デフォルトは `#pragma pack(8)` です。

例

この例では、`pack(2)` によって整数変数 `b` が 2 バイト境界で整列される方法を示します。

```
typedef struct
{
    char a;
    int b;
} S;
#pragma pack(2)
typedef struct
{
    char a;
    int b;
} SP;
S var = { 0x11, 0x44444444 };
SP pvar = { 0x11, 0x44444444 };
```

`S` のレイアウトは以下のとおりです。

0	1	2	3
a	padding		
4	5	6	7
b	b	b	b

図 4-1 非パック構造体 S

`SP` のレイアウトは以下のとおりです。

0	1	2	3
a	x	b	b
4	5		
b	b		

図 4-2 パック構造体 SP

——— 注 ———

このレイアウトでは、`x` が 1 バイトのパディングを示します。

`SP` は 6 バイト構造体です。`b` の後にパディングはありません。

———————————

4.4 #pragma unroll[(n)], #pragma unroll_completely

コンパイラに *n* 回の繰り返しによるループの展開を指示します。

構文

```
#pragma unroll
#pragma unroll_completely
#pragma unroll n
#pragma unroll( n )
```

各項目には以下の意味があります。

n
展開のための繰り返し回数を示すオプション値です。

デフォルト

n の値を指定しないと、コンパイラはループを完全に展開しようとします。コンパイラが繰り返し回数を判別できる場合にのみ、ループを完全に展開できます。

`#pragma unroll_completely` は繰り返し回数が指定されていない `#pragma unroll` と同じ意味です。

使用法

このプラグマは、最適化レベル `-O2` 以上の場合にのみ機能します。

`-O3` を使用してコンパイルする場合、ループの展開が有効な場所では自動的にループが展開されます。このプラグマを使用すると、自動的に展開されないループの展開をコンパイラに要求できます。

`#pragma unroll[(n)]` は、 **for** ループ、 **while** ループ、または **do** ... **while** ループの直前でのみ使用できます。

制約条件

このプラグマは、自動的に展開されなかったループを展開するための、コンパイラへの*要求*です。必ずしもループが展開されるとは限りません。

4.5 #pragma weak symbol、#pragma weak symbol1 = symbol2

このプラグマは、シンボルを weak としてマークしたり、シンボルの weak エイリアスを定義したりするための言語拡張機能です。

例

以下の例では、`__weak_fn` の weak エイリアスとして `weak_fn` が宣言されています。

```
extern void weak_fn(int a);
#pragma weak weak_fn = __weak_fn
void __weak_fn(int a)
{
    ...
}
```

第 5 章
その他のコンパイラ固有の機能

事前定義のマクロなど、C および C++ 標準の拡張機能であるコンパイラ固有の機能の概要を示します。

以下のセクションから構成されています。

5.1 定義済みマクロ

ARM コンパイラによって、いくつかのマクロが事前定義されます。これらのマクロは、ツールチェーンのバージョン番号やコンパイラオプションについての情報を提供します。

一般に、コンパイラによって生成された事前定義のマクロは、GCC によって生成されたものに対応しています。詳細については、GCC ドキュメントをご覧下さい。

次の表に、C および C++ 用の ARM コンパイラにより事前定義された ARM 固有のマクロの名前と、最もよく使用されているさまざまなマクロの名前を示します。値フィールドに記載がない場合は、シンボルが定義されているだけです。

注

`armclang --target=`*`triple`* `-E -dM` *ファイル* を使用して、事前定義のマクロの値を確認できます。

表 5-1 定義済みマクロ

Name	値	定義されるケース
__ARM_64BIT_STATE	1	64 ビットターゲットのみのセットです。 コードが 64 ビットステートの場合、1 に設定します。
__ARM_ALIGN_MAX_STACK_PWR	4	64 ビットターゲットのみのセットです。 スタックオブジェクトの最大境界整列のログ。
__ARM_ARCH	*ver*	8 など、ターゲットアーキテクチャのバージョンを指定します。
__ARM_ARCH_EXT_IDIV__	1	32 ビットターゲットのみのセットです。 ハードウェア除算命令が使用可能な場合は 1 に設定します。
__ARM_ARCH_ISA_A64	1	64 ビットターゲットのみのセットです。 ターゲットが A64 命令セットをサポートしている場合は 1 に設定します。
__ARM_ARCH_PROFILE	*ver*	A など、ターゲットアーキテクチャのプロファイルを指定します。
__ARM_FEATURE_CLZ	1	64 ビットターゲットのみのセットです。 ハードウェアで CLZ（先行ゼロカウント）命令がサポートされている場合は 1 に設定します。
__ARM_FEATURE_DIV	1	64 ビットターゲットのみのセットです。 ターゲットが結合浮動小数点積和をサポートしている場合は 1 に設定します。
__ARM_FEATURE_CRC32	1	32 ビットターゲットのみのセットです。 ターゲットに、CRC 拡張機能がある場合は、1 に設定されます。
__ARM_FEATURE_CRYPTO	1	ターゲットに、暗号化拡張機能がある場合は、1 に設定されます。
__ARM_FEATURE_FMA	1	64 ビットターゲットのみのセットです。 ターゲットが結合浮動小数点積和をサポートしている場合は 1 に設定します。
__ARM_FEATURE_UNALIGNED	1	64 ビットターゲットのみのセットです。 ハードウェアでターゲットに非境界整列アクセスがある場合、1 に設定します。

表 5-1 定義済みマクロ（続き）

Name	値	定義されるケース
__ARM_FP	0xE	64 ビットターゲットのみのセットです。 ハードウェア浮動小数点が使用できる場合は設定します。
__ARM_FP_FAST	1	64 ビットターゲットのみのセットです。 -ffast-math が指定されている場合は設定します。
__ARM_FP_FENV_ROUNDING	1	64 ビットターゲットのみのセットです。 実装により、標準 C fesetround() 関数を使用して実行時に丸めを設定できる場合、1 に設定します。
__ARM_NEON__	-	コンパイラが Advanced SIMD が使用できるアーキテクチャまたはプロセッサをターゲットとしている場合に定義されます。 Advanced SIMD コンパイラ組み込み関数の使用を許可する際、このマクロを使用すると、arm_neon.h を条件付きでインクルードすることができます。
__ARM_NEON_FP	7	64 ビットターゲットのみのセットです。 Advanced SIMD 浮動小数点ベクトル命令が使用できる場合に設定します。
__ARM_PCS	1	32 ビットターゲットのみのセットです。 デフォルトの変換単位のプロシージャコール標準がベース PCS に準拠している場合に 1 に設定します。
__ARM_PCS_VFP	1	32 ビットターゲットのみのセットです。 -mfloat-abi=hard の場合に 1 に設定します。
__ARM_SIZEOF_MINIMAL_ENUM	*value*	64 ビットターゲットのみのセットです。 最小列挙型のサイズを指定します。-fshort-enums が指定されているかどうかによって、1 か 4 のいずれかに設定します。
__ARMCOMPILER_VERSION	*nnnbbbb*	常に、コンパイラ armclang のバージョン数を指定します。 フォーマット *nnnbbbb* です（ *nnn* はバージョン番号で *bbbb* はビルド番号）。 例えば、バージョン 6.0 ビルド 0654 は 6000654 と表します。
__ARMCC_VERSION	*nnnbbbb*	__ARMCOMPILER_VERSION と同じ意味。
__arm__	1	--target=arm-arm-none-eabi などの、AArch32 ターゲットを持つ A32 または T32 命令セットをターゲットとする場合に定義されます。 __aarch64__ も参照して下さい。
__aarch64__	1	--target=aarch64-arm-none-eabi をもつ A64 命令セットをターゲットとする場合に定義されます。 __arm__ も参照して下さい。

表 5-1 定義済みマクロ（続き）

Name	値	定義されるケース
__cplusplus	*ver*	C++ コードをコンパイルする場合に定義され、ターゲットとされる C++ 標準を識別する値に設定されます。たとえば、`-xc++ -std=gnu++98` を使用してコンパイルする場合、コンパイラはこのマクロを `199711L` に設定します。 `__cplusplus` マクロを使用すると、C コンパイラまたは C++ コンパイラがファイルをコンパイルしたかどうかをテストすることができます。
__CHAR_UNSIGNED__	1	**char** が符号なしの型である場合にのみ定義されます。
__EXCEPTIONS	1	例外がイネーブルになった状態で C++ ソースファイルをコンパイルする場合に定義されます。
__GNUC__	*ver*	常に、互換 GCC バージョンの現在のメジャーバージョンを示す整数値が設定されます。
__GNUC_MINOR__	*ver*	常に、互換 GCC バージョンの現在のマイナーバージョンを示す整数値が設定されます。
__INTMAX_TYPE__	*タイプ*	常に、**intmax_t typedef** の正しい潜在型を定義します。
__NO_INLINE__	1	関数がインライン化されていない場合に定義されます。マクロは常に、最適化レベル `-O0` または `-fno-inline` オプションが指定される場合に定義されます。
__OPTIMIZE__	1	`-O1`、`-O2`、`-O3`、`-Ofast`、`-Oz`、または `-Os` が指定される場合に定義されます。
__OPTIMIZE_SIZE__	1	`-Os` または `-Oz` が指定される場合に定義されます。
__PTRDIFF_TYPE__	*タイプ*	常に、**ptrdiff_t typedef** の正しい潜在型を定義します。
__SIZE_TYPE__	*タイプ*	常に、**size_t typedef** の正しい潜在型を定義します。
__SOFTFP__	1	32 ビットターゲットのみのセットです。 それ以外の場合は、0 に設定されます。
__STDC__	1	常に、コンパイラが ISO 標準規格 C に準拠していることを示します。
__STRICT_ANSI__	1	`--ansi` オプションを指定する場合または `--std=c*` オプションの 1 つを指定する場合に定義されます。
__thumb__	1	`-mthumb` オプションを指定する場合に定義されます。
__UINTMAX_TYPE__	*タイプ*	常に、**uintmax_t typedef** の正しい潜在型を定義します。
__VERSION__	*ver*	常に、ベースとなっている Clang バージョンを示す文字列。
__WCHAR_TYPE__	*タイプ*	常に、**wchar_t typedef** の正しい潜在型を定義します。
__WINT_TYPE__	*タイプ*	常に、**wint_t typedef** の正しい潜在型を定義します。

関連参照

1.41 --version_number（1-56 ページ）.
1.35 -std（1-50 ページ）.
1.32 -O（1-47 ページ）.
1.37 --target（1-52 ページ）.
1.24 -marm（1-38 ページ）.
1.30 -mthumb（1-45 ページ）.

5.2 インライン関数

インライン関数を使用すると、コードサイズとパフォーマンスのトレードオフをとることができます。デフォルトで、コードをインライン展開すべきかどうかは、コンパイラによって自動的に決定されます。

一般に、`-Ospace` を使用してコンパイルする場合、コンパイラは、最小サイズのコードを生成するという観点から、インライン展開に関して適切な判断を下します。`-Otime` を使用してコンパイルする場合、ほとんどのケースでインライン展開が行われる一方で、コードの増大化も回避されます。

ほとんどの場合、特定の関数をインライン展開すべきかどうかの判断は、コンパイラに任せるのが賢明です。関数を **`__inline__`** または **`inline`** キーワードで修飾すると、コンパイラに対してその関数をインライン展開するように指示しますが、最終的な判断はコンパイラが下します。関数を **`__attribute((always_inline))`** で修飾すると、関数をインライン展開するようにコンパイラに強制します。

リンカは非常に短い関数に対してある程度の関数のインライン展開を適用できます。

注

C ソースコードのデフォルトの意味上の規則は、C99 規則に従います。インライン展開では、関数のインライン展開が指示されると、コンパイラはインライン展開しない方がよいと判断した時点で使用する別の非修飾の関数のバージョンをコード内で見つけようとします。コンパイラが非修飾バージョンを見つけられない場合には、以下のエラーが表示され失敗に終わります。

`"Error: L6218E: Undefined symbol <symbol> (referred from <file>)"`.

この問題を回避するために、以下のようなオプションがあります。

- 同等の関数の非修飾バージョンを提供する。
- 修飾子を **`static inline`** に変更する。
- 提案としての役割しか果たさないため、 **`inline`** キーワードを削除する。
- `-std=gnu90` オプションを使用して、GNU C90 言語でプログラムをコンパイルする。

関連参照

2.8 __inline（2-71 ページ）.
1.35 -std（1-50 ページ）.

関連情報

__attribute((always_inline)).