

# CodeWarrior®

## C Compilers Reference



CodeWarrior に関する最新のテクニカル情報については CodeWarriorCD の Release Notes フォルダをご覧ください。

Revised: 99/12/06 map-JP000323

Metrowerks CodeWarrior © Copyright 1993-2000 by Metrowerks Inc. and its Licensors. All rights reserved.

お客様は、本 CD に記録されている文書を個人使用目的に限り、プリントすることができます。この場合を除いて、Metrowerks Inc. からの書面による承諾なしに、本 CD に記録されている文書の全部、または、一部をいかなる形態、方法（電子的、物理的な複製、または、写真複写、録音録画、その他すべての情報記録、再生システムを含む）により、複製または伝達することを禁じます。

Metrowerks の名称、ロゴ、CodeWarrior、PowerPlant、Metrowerks University は、Metrowerks Inc. の登録商標です。

Geekware、Discover Programming は、Metrowerks Inc. の商標です。

記載の商標および登録商標は、各社が保有します。

CD に記録されているすべてのソフトウェアおよび文書は、CodeWarrior QuickStart の巻末に記述されているライセンス契約が適用されます。

連絡先：

---

日本	メトロワークス株式会社 150-0042 東京都渋谷区宇田川町 36-6 ワールド宇田川ビル 8F TEL : (03) 3780-6091      FAX : (03) 3780-6092
U.S.A.	Metrowerks Corporation 9801 Metric Boulevard, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
WWW サーバ	<a href="http://www.metrowerks.co.jp/">http://www.metrowerks.co.jp/</a> <a href="http://www.metrowerks.com/">http://www.metrowerks.com/</a>
ユーザー登録	<a href="mailto:j-register@metrowerks.com">j-register@metrowerks.com</a>
テクニカルサポート	<a href="mailto:j-support@metrowerks.com">j-support@metrowerks.com</a>
購入 / 契約更新	<a href="mailto:j-sales@metrowerks.com">j-sales@metrowerks.com</a>
インフォメーション	<a href="mailto:j-info@metrowerks.com">j-info@metrowerks.com</a>

---

# 目次

---

第 1 章 紹介	11
このマニュアルについて	11
リリースノートについて	11
新機能	12
マニュアルの表記規則	12
文法について	12
表記について	12
第 2 章 C/C++ コンパイラの設定	15
コンパイラ設定の概要	15
C/C++ 言語設定パネル	15
C/C++ 警告設定パネル	18
警告をエラーとしてあつかう	20
不当な Pragma	20
識別子がない宣言	20
間違いやすいエラー	20
未使用変数	21
未使用引数	22
余分なコンマ	23
拡張エラーチェック	23
仮想関数が隠ぺいされた場合	24
暗黙の算術変換	24
関数がインライン展開されなかった場合	25
キーワード 'class' と 'struct' の一貫性のない使用	25
第 3 章 C コンパイラ	27
C コンパイラの概要	27
C の実装	27
識別子	28
インクルードファイル	28
プリフィックスファイル	30
Sizeof() オペレータ	30
Volatile 変数	30
enum は常に Int 型	31
ANSI/ISO C の拡張	32
ANSI に厳格に従う	33
wchar_t 型サポート	34
C++ 形式のコメント	34
関数内で名前のない引数	34
マクロ定義内で引数を伴わない #	35

#endif 後の識別子	35
キャストされたポインタを lvalue として使用	36
アドレスによる変数宣言	36
ANSI キーワードのみ	36
Trigraph 拡張	37
整数値としてのキャラクタ定数	37
関数のインライン	38
マルチバイト文字（日本語など）に対応	39
文字列定数を一カ所にまとめる	39
文字列定数を再利用しない	40
関数プロトタイプが必要	41
CR の代わりに NL を利用	42
ポインタタイプルールを緩める	43
unsigned char を使用	43
64 ビット整数を使う	43
ポインタを同じサイズの型へ変換する	44
コンパイル時にアラインメントと型情報を取得	44
構造体内のゼロ長の配列	44
ビット回転の組込み関数	45
D 定数接尾子	45
short double 型	45
GNU C 拡張	45

---

## 第 4 章 C++ コンパイラ 47

C++ コンパイラの概要	47
C++ の実装	48
main() の暗黙のリターンステートメント	48
キーワード順	48
追加キーワード	49
条件オペレータの規約	49
メンバー関数のデフォルト引数	49
インライン関数のローカルクラス宣言	50
クラスオブジェクトのコピーと構築	50
静的データの初期化用リソースチェック	51
継承されたメンバー関数の呼び出し	51
サポートされていない拡張	53
C++ コンパイラの設定	53
C++ コンパイラを必ず利用	54
ARM に適合	54
C++ 例外処理有効	55
RTTI 有効	55
bool 型サポート	55

C++ 拡張の追加	56
C++ 例外処理の使用	57
RTTI の使用	57
dynamic_cast オペレータの使用	57
typeid オペレータの使用	59
テンプレートの使用	60
テンプレートの宣言と定義	60
テンプレートのインスタンス化	62
<b>第 5 章 C++ とエンベデッドシステム</b>	<b>65</b>
C++ とエンベデッドシステムの概要	65
EC++ 互換モード	65
ANSI/ISO C++ と EC++ の違い	65
テンプレート	66
ライブラリ	66
ファイル操作	66
ローカリゼーション	66
例外処理	66
その他の言語特性	66
CodeWarrior の EC++ 仕様	66
言語関係の問題	67
ライブラリ関係の問題	67
C++ コードのサイズ	67
サイズの最適化	68
インライン化	68
仮想関数	69
RTTI	69
例外処理	69
オペレータ new	69
多重継承	70
仮想継承	70
ストリームベースクラス	70
他のクラスライブラリを使う	70
<b>第 6 章 プラグマとシンボル</b>	<b>71</b>
プラグマとシンボルの概要	71
プラグマ	71
プラグマの文法	72
プラグマの有効範囲	72
a6frames	73
align	74
align_array_members	74

altivec_codegen . . . . .	.75
altivec_model . . . . .	.76
altivec_vrsave. . . . .	.76
always_inline . . . . .	.76
ANSI_strict . . . . .	.77
arg_dep_lookup . . . . .	.78
ARM_conform . . . . .	.78
auto_inline . . . . .	.79
bool . . . . .	.80
check_header_flags . . . . .	.80
code_seg . . . . .	.80
code68020 . . . . .	.81
code68881 . . . . .	.82
cplusplus . . . . .	.82
cpp_extensions . . . . .	.83
d0_pointers . . . . .	.84
data_seg . . . . .	.85
def_inherited . . . . .	.85
defer_codegen . . . . .	.85
define_section. . . . .	.86
direct_destruction . . . . .	.90
direct_to_som. . . . .	.90
disable_registers . . . . .	.90
dollar_identifiers . . . . .	.91
dont_inline . . . . .	.92
dont_reuse_strings. . . . .	.92
ecplusplus . . . . .	.93
EIPC_EIPSW. . . . .	.93
enumsalwaysint . . . . .	.93
exceptions . . . . .	.94
export . . . . .	.95
extended_errorcheck . . . . .	.96
far_code, near_code, smart_code . . . . .	.97
far_data . . . . .	.98
far_strings . . . . .	.98
far_vtables . . . . .	.99
faster_pch_gen . . . . .	.99
float_constants . . . . .	100
force_active . . . . .	100
fourbyteints . . . . .	101
fp_contract . . . . .	101
fp_pilot_traps. . . . .	102
fullpath_prepdump . . . . .	102

function . . . . .	103
gcc_extensions . . . . .	103
global_optimizer, optimization_level . . . . .	103
IEEEdoubles . . . . .	104
ignore_oldstyle . . . . .	104
import . . . . .	105
init_seg . . . . .	106
inline_depth . . . . .	107
inline_intrinsics . . . . .	107
internal . . . . .	107
interrupt. . . . .	109
interrupt_fast . . . . .	109
k63d . . . . .	109
k63d_calls . . . . .	110
lib_export . . . . .	110
line_prepdump . . . . .	111
longlong . . . . .	111
longlong_enums . . . . .	111
longlong_prepval . . . . .	112
macsbug, oldstyle_symbols . . . . .	112
mark . . . . .	113
message. . . . .	114
microsoft_exceptions. . . . .	114
microsoft_RTTI . . . . .	114
mmx . . . . .	115
mmx_call . . . . .	115
mpwc . . . . .	115
mpwc_newline . . . . .	116
mpwc_relax . . . . .	117
no_register_coloring . . . . .	118
no_static_dtors . . . . .	119
once . . . . .	119
only_std_keywords . . . . .	119
opt_common_subs . . . . .	120
opt_dead_assignments . . . . .	120
opt_dead_code . . . . .	121
opt_lifetimes . . . . .	121
opt_loop_invariants . . . . .	121
opt_propagation . . . . .	122
opt_strength_reduction . . . . .	122
opt_unroll_loops. . . . .	122
opt_vectorize_loops . . . . .	123
optimization_level . . . . .	123

---

optimize_for_size . . . . .	123
oldstyle_symbols . . . . .	124
pack . . . . .	124
parameter. . . . .	125
pcrelstrings . . . . .	125
peephole . . . . .	126
ppc_unroll_factor_limit . . . . .	127
ppc_unroll_instructions_limit . . . . .	127
ppc_unroll_speculative. . . . .	127
pointers_in_A0, pointers_in_D0 . . . . .	128
pool_data. . . . .	129
pool_strings . . . . .	129
pop, push . . . . .	130
precompile_target . . . . .	130
profile . . . . .	131
readonly_strings . . . . .	132
register_coloring . . . . .	132
require_prototypes. . . . .	133
実行時型情報 ( RTTI ) . . . . .	133
schedule . . . . .	134
scheduling . . . . .	134
section . . . . .	135
segment . . . . .	139
side_effects . . . . .	140
simple_prepdump . . . . .	141
SOMCallOptimization. . . . .	141
SOMCallStyle . . . . .	142
SOMCheckEnvironment . . . . .	142
SOMClassVersion. . . . .	144
SOMMetaClass . . . . .	144
SOMReleaseOrder . . . . .	145
stack_cleanup. . . . .	145
static_inlines . . . . .	146
suppress_init_code . . . . .	146
sym . . . . .	147
syspath_once . . . . .	147
toc_data . . . . .	148
trigraphs . . . . .	148
traceback. . . . .	149
unsigned_char. . . . .	149
unused . . . . .	149
use_fp_instructions . . . . .	150
use_frame . . . . .	150



use_mask_registers . . . . .	151
warn_emptydecl . . . . .	151
warning_errors . . . . .	152
warn_extracomma . . . . .	152
warn_hidevirtual . . . . .	152
warn_illpragma . . . . .	153
warn_implicitconv . . . . .	154
warn_no_side_effect . . . . .	154
warn_notinlined . . . . .	155
warn_padding . . . . .	155
warn_possunwant . . . . .	155
warn_resultnotused . . . . .	156
warn_structclass . . . . .	157
warn_unusedarg . . . . .	157
warn_unusedvar . . . . .	158
warning . . . . .	159
wchar_type . . . . .	159
定義済みシンボル . . . . .	160
ANSI の定義済みシンボル . . . . .	160
Metrowerks の定義済みシンボル . . . . .	161
オプションのチェック . . . . .	162



# 第 1 章 紹介

このマニュアルでは CodeWarrior C/C++ コンパイラの使い方を説明します。各章はターゲットに共通する C/C++ コンパイラの情報、各ターゲットやオペレーティングシステムに特有の情報に分けられています。

[このマニュアルについて](#)

[リリースノートについて](#)

[新機能](#)

[マニュアルの表記規則](#)

## このマニュアルについて

このマニュアルでは以下の内容について説明します。

インターフェース：コンパイラのオプションの設定方法を説明します。

言語：すべての CW ターゲットに共通するコンパイラの情報説明します。

プラグマ：すべてのターゲット用のプラグマの情報を説明します。

各章の最初には概要があります。表 1.1 にこのマニュアルの内容をまとめます。

表 1.1 このマニュアルの内容

章	内容
<a href="#">コンパイラ設定の概要</a>	C/C++ 言語設定パネル、C/C++ 警告設定パネルについての参照ページの紹介
<a href="#">C コンパイラの概要</a>	CodeWarrior C/C++ コンパイラの C 言語の実装
<a href="#">C++ コンパイラの概要</a>	CodeWarrior C/C++ コンパイラの C++ 特有の機能
<a href="#">C++ とエンベデッドシステムの概要</a>	CodeWarrior C++ でのエンベデッドシステム開発と EC++ ( Embedded C++ ) 規格の情報
<a href="#">プラグマとシンボルの概要</a>	CodeWarrior C/C++ コンパイラで利用できるプラグマとシンボル

## リリースノートについて

CodeWarrior C/C++ コンパイラに変更が加えられ、このマニュアルでの情報と実際の機能が異なる可能性があります。最新情報は CodeWarrior CD のリリースノートに含まれていますのでお読みください。

## 新機能

これはバージョン 2.3 の CodeWarrior C/C++ のマニュアルです。

今回のバージョンで改訂、追加された部分を紹介します。

[「short double 型」\(p45\)](#): Mac OS プログラミングにおける非標準データ型の使用

[「関数のインライン」\(p38\)](#): 遅延インラインについて

[「GNU C 拡張」\(p45\)](#): CodeWarrior C の、GNU C 言語拡張へのサポート

[「プラグマとシンボル」\(p71\)](#): 新しいプラグマの説明 ([altivec\\_codegen](#)、[altivec\\_model](#)、[altivec\\_vrsave](#)、[fullpath\\_prepdump](#)、[gcc\\_extensions](#)、[line\\_prepdump](#)、[ppc\\_unroll\\_factor\\_limit](#)、[ppc\\_unroll\\_factor\\_limit](#)、[ppc\\_unroll\\_speculative](#)、[warn\\_no\\_side\\_effect](#)、[warn\\_resultnotused](#))

## マニュアルの表記規則

ここではマニュアルの表記規則を説明します。

[文法について](#)

[表記について](#)

### 文法について

このマニュアルには以下のようなステートメントの文法例が記載されています。

```
#pragma parameter [return-reg] func-name [param-regs]
#pragma optimize_for_size on | off | reset
```

[表 1.2](#) に、これらのステートメントの解釈についてまとめます。

表 1.2 文法の例

テキスト	意味
literal	それをそのままソースコードに書く
<i>metasympol</i>	そのシンボルを適切な値に置き換える。文法例の後に適切な値について記載
a b c	a、b、c のいずれか 1 つを使う
[a]	必要ならばこのシンボルを使う。文法例の後で必要な場合を説明。

### 表記について

特定の情報を表すスタイルについて説明します。

注意、警告、ヒント、および初心者用のヒント

「注意」は、重要な事実を言い換えたり、自明ではない事実に注意を向けます。

「警告」は、実行すると取り返しのつかないものを注意したり、発生する可能性のあるエラーを知らせます。

「ヒント」は、CodeWarrior をより活用するためのヒントです。

「初心者」は、プログラミングの初心者が用語や概念をよりよく理解できるようにします。

#### 書体の規則

異なる書体 (Courier という書体です) のテキストは、ファイル名やフォルダ名、コード、またはコンピュータのハードディスク上で見られるその他の項目を示します。

CodeWarrior のメニューにある項目は括弧付き ( [ 開く ] など ) で示します。

下線付きの青色のテキストは ( 例 : [「IDE User Guide の概要」\(p17\)](#) ) オンラインドキュメント ( Adobe Acrobat など ) でのハイパーテキストを示します。これをクリックすると該当ページへジャンプします。



## 第 2 章 C/C++ コンパイラの設定

ここでは C/C++ 言語設定パネル、C/C++ 警告設定パネルの参照ページを紹介します。

### コンパイラ設定の概要

[「C/C++ 言語設定パネル」\(p15\)](#)：設定可能なオプションとその参照ページの紹介

[「C/C++ 警告設定パネル」\(p18\)](#)：コンパイラの警告とその参照ページの紹介

C/C++ 言語設定パネルは、C/C++ コンパイラ設定パネルとも呼ばれます。

### C/C++ 言語設定パネル

C/C++ 言語設定パネル ([図 2.1](#)) のオプションで、C/C++ コンパイラの挙動を設定します。その他の設定パネルについては『IDE User Guide』を参照してください。

---

ヒント：ソースコードでプラグマ疑似命令を使うことでもオプションの設定が可能です。詳細は [「プラグマとシンボル」\(p71\)](#) を参照してください。

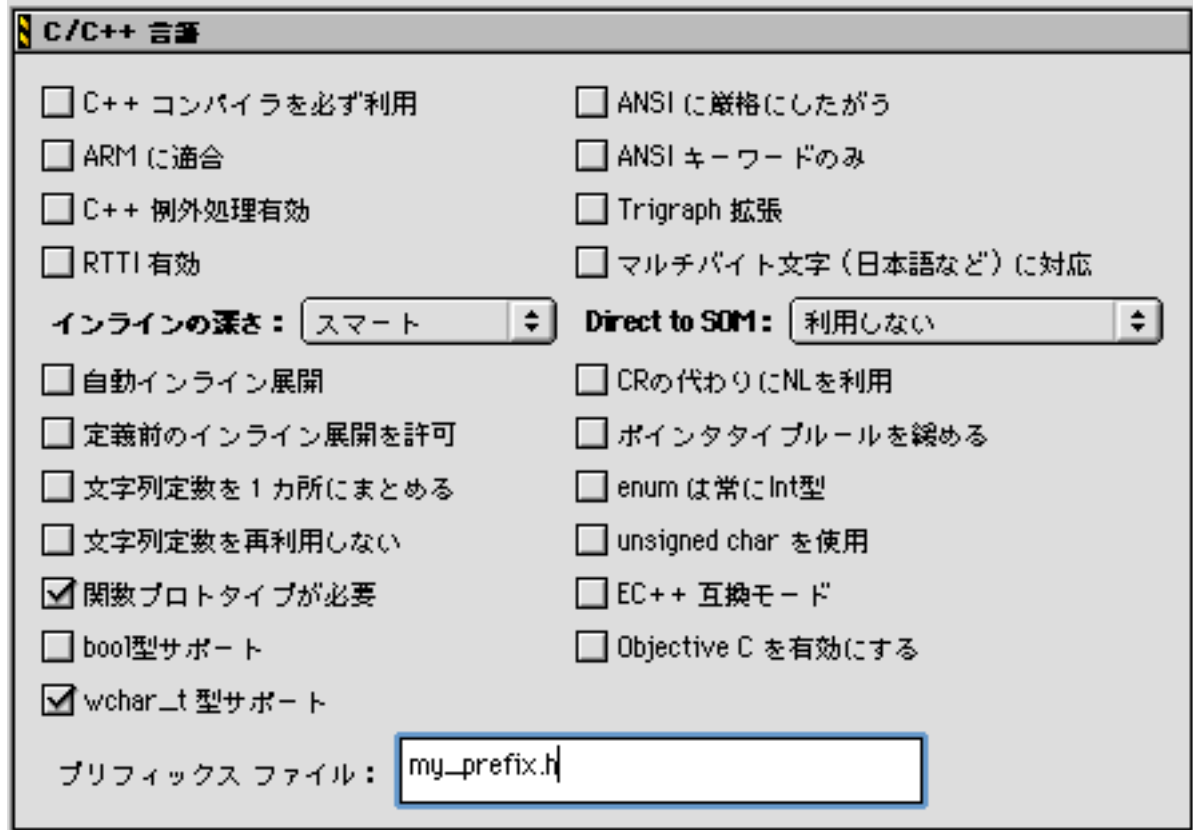
---

コンパイラのオプションにはそれぞれ相当するプラグマがあります。あるオプションをオンまたはオフに設定するには、設定パネルのチェックボックスをクリックするか、メニューから選択します。ソースコードでプラグマを使うと、C/C++ 言語設定パネルでのオプションの設定に関係なく、そのオプションをオンまたはオフにできます。C/C++ 言語設定パネルのオプションに相当しないプラグマもいくつかあります。詳しくは [「プラグマとシンボルの概要」\(p71\)](#) を参照してください。

コードで特殊なプリプロセッサ疑似命令を使って、各オプションの現在の設定を調べることもできます。これらの疑似命令については [「オプションのチェック」\(p162\)](#) を参照してください。

[図 2.1](#) のオプションには、ターゲットによっては表示されないものもあります。例えば [ Direct to SOM ] ポップアップメニューは Mac OS ターゲットでのみ表示されます。

図 2.1 C/C++ 言語設定パネル



設定パネルの各オプションについては他の章で詳しく説明します。コンパイラの C/C++ 規格の実装方法に密接な関係があるからです。

その他のオプション

以下の表に、C/C++ 言語設定パネルにあるオプションについての参照ページをまとめます。

オプション名	参照ページ
C++ コンパイラを必ず利用 ( Activate C++ Compiler )	<a href="#">「C++ コンパイラを必ず利用」( p54 )</a>
ARM に適合 ( ARM Conformance )	<a href="#">「ARM に適合」( p54 )</a>
C++ 例外処理有効 ( Enable C++ Exceptions )	<a href="#">「C++ 例外処理有効」( p55 )</a>
RTTI 有効 ( Enable RTTI )	<a href="#">「RTTI 有効」( p55 )</a>



オプション名	参照ページ
インラインの深さ (Inline Depth) 自動インライン展開 (Auto-inline) 定義前のインライン展開を許可 (Deferred Inlinings)	<a href="#">「関数のインライン」(p38)</a>
文字列定数を一ヶ所にまとめる (Pool Strings)	<a href="#">「文字列定数を一ヶ所にまとめる」(p39)</a>
文字列定数を再利用しない (Don't Reuse Strings)	<a href="#">「文字列定数を再利用しない」(p40)</a>
関数プロトタイプが必要 (Require Function Prototypes)	<a href="#">「関数プロトタイプが必要」(p41)</a>
bool 型サポート (Enable bool Support)	<a href="#">「bool 型サポート」(p55)</a>
wchar_t 型サポート (Enable wchar_t Support)	<a href="#">「wchar_t 型サポート」(p34)</a>
ANSI に厳格に従う (ANSI Strict)	<a href="#">「ANSI に厳格に従う」(p33)</a>
ANSI キーワードのみ (ANSI Keywords Only)	<a href="#">「ANSI キーワードのみ」(p36)</a>
Trigraphs 拡張 (Expand Trigraphs)	<a href="#">「Trigraph 拡張」(p37)</a>
マルチバイト文字 (日本語など) に対応 (Multi-Byte Aware)	<a href="#">「マルチバイト文字 (日本語など) に対応」(p39)</a>
Direct to SOM	『Targeting Mac OS』マニュアル
CR の代わりに NL を利用 (Map Newlines to CR)	<a href="#">「CR の代わりに NL を利用」(p42)</a>
ポインタタイプルールを緩める (Relaxed Pointer Type Rules)	<a href="#">「ポインタタイプルールを緩める」(p43)</a>
Enum は常に Int 型 (Enums Always Int)	<a href="#">「enum は常に Int 型」(p31)</a>
Unsigned Char を使用 (Use Unsigned Chars)	<a href="#">「unsigned char を使用」(p43)</a>
Prefix File (プリフィックスファイル)	<a href="#">「プリフィックスファイル」(p30)</a>
EC++ 互換モード (EC++ Compatibility Mode)	<a href="#">「EC++ 互換モード」(p65)</a>
Objective C を有効にする (Enable Objective C)	『Targeting Mac OS X』マニュアル

## C/C++ 警告設定パネル

C/C++ コンパイラは、不当な文法など解析できないコードを発見するとエラーを発生します。コンパイラはエラーだけでなく、不当ではないが意図的ではない文法を知らせる警告メッセージも表示します。これらの間違いは C/C++ 言語としては正当ですが、予想通りには動作しません。

コンパイラはこれらの間違いを発見すると警告を出します。警告は致命的なものではありません。警告をエラーとして扱うと、コードをコンパイルすることはできますが、正確には動作しません。

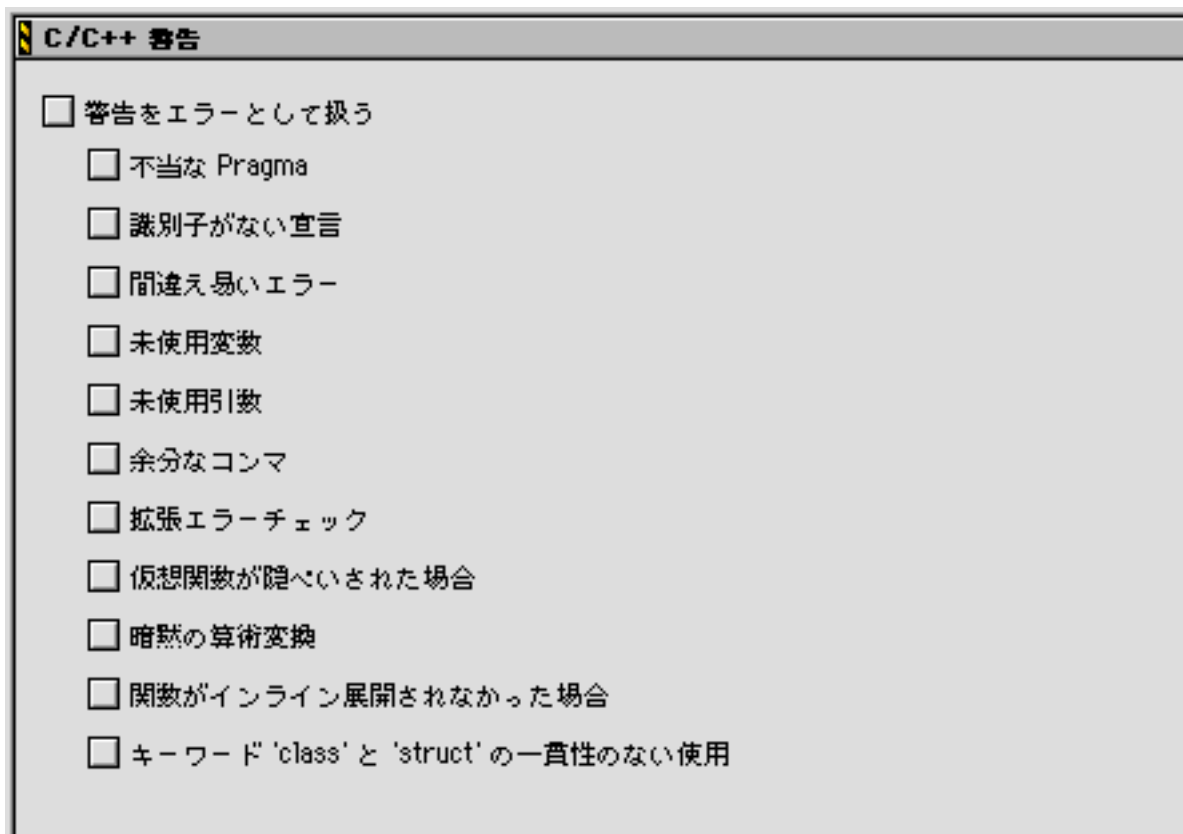
ここでは C/C++ 警告設定パネル ([図 2.2](#)) のオプションについて説明します。オプションの設定によって、どの警告を表示するのかを決定します。

警告を表示するか否かを設定するには、そのチェックボックスをクリックしてオンまたはオフにします。警告にはそれぞれ相当するプラグマがあります。ソースコードでプラグマを使うと、コードの一部に対して特定の警告を表示または非表示に設定することができます。使用可能なプラグマについての詳細は「[コンパイラ設定の概要](#)」(p15) を参照してください。

コードで特殊なプリプロセッサ疑似命令を使って、各オプションの現在の設定を調べることもできます。これらの疑似命令については「[オプションのチェック](#)」(p162) を参照してください。

その他の設定パネルについては『IDE User Guide』を参照してください。

図 2.2 C/C++ 警告設定パネル



ここでは以下の項目について警告を出すオプションを説明します。

[「警告をエラーとしてあつかう」\(p20\)](#) ( Treat All Warnings As Errors )

[「不当な Pragma」\(p20\)](#) ( Illegal Pragmas )

[「識別子がない宣言」\(p20\)](#) ( Empty Declarations )

[「間違えやすいエラー」\(p20\)](#) ( Possible Errors )

[「未使用変数」\(p21\)](#) ( Unused Variables )

[「未使用引数」\(p22\)](#) ( Unused Arguments )

[「余分なコンマ」\(p23\)](#) ( Extra Commas )

[「拡張エラーチェック」\(p23\)](#) ( Extended Error Checking )

[「仮想関数が隠ぺいされた場合」\(p24\)](#) ( Hidden Virtual Functions )

[「暗黙の算術変換」\(p24\)](#) ( Implicit Arithmetic Conversions )

[「関数がインライン展開されなかった場合」\(p25\)](#) ( Non-Inlined Functions )

[「キーワード 'class' と 'struct' の一貫性のない使用」\(p25\)](#) ( Inconsistent Use of 'class' and 'struct' Keywords )

## 警告をエラーとしてあつかう

C/C++ 警告設定パネルの [ 警告をエラーとして扱う ] オプションがオンの場合、コンパイラはすべての警告をエラーとして扱います。すべての警告が解決されるまで、コンパイルはできません。

[ 警告をエラーとして扱う ] オプションは、`warning_errors` に相当します (「[warning\\_errors](#)」(p152))。このプラグマが `on` か否かをチェックするには、`__option (warning_errors)` を使います (「[オプションのチェック](#)」(p162))。

## 不当な Pragma

[ 不当な Pragma ] オプションがオンの場合、コンパイラは不当なプラグマを検出すると警告を出します。例えば、以下のプラグマは警告になります。

---

```
#pragma near_data off      // WARNING: near_dat はプラグマではない
#pragma far_data select    // WARNING: select は定義されていない
#pragma far_data on        // OK
```

---

[ 不当な Pragma ] オプションは、`warn_illpragma` に相当します (「[warn\\_illpragma](#)」(p153))。このプラグマが `on` か否かをチェックするには、`__option (warn_illpragma)` を使ってください (「[オプションのチェック](#)」(p162))。

## 識別子がない宣言

[ 識別子がない宣言 ] オプションがオンの場合、コンパイラは変数のない宣言を見つけると警告を出します。

---

```
int ;                      // WARNING
int i;                     // OK
```

---

[ 識別子がない宣言 ] オプションは、`warn_emptydecl` に相当します (「[warn\\_emptydecl](#)」(p151))。このプラグマが `on` か否かをチェックするには、`__option (warn_emptydecl)` (「[オプションのチェック](#)」(p162)) を使ってください。

## 間違えやすいエラー

[ 間違え易いエラー ] オプションがオンの場合、コンパイラは、C/C++ に則っているけれど予期せぬ副次効果を起こす可能性がある入力ミスをチェックします (例えば、意図的でないセミコロンの挿入や、`=` と `==` の間違いなど)。コンパイラは、以下のいずれかを見つけると警告を出します。

`if`、`while`、`for` の条件内に代入がある場合。このチェックは、`==` を使う時に `=` を使いがちな人にとって有効です。以下に例を示します。

---

```
if (a=b) f();           // WARNING: a=b は代入
if ((a=b)!=0) f();      // OK: (a=b)!=0 は比較
if (a==b) f();          // OK: (a==b) は比較
```

---

1つの表記内に等値比較がある場合。このチェックは、=を使う時に==を使いがちな人にとって有効です。以下に例を示します。

---

```
a == 0;                // WARNING: これは比較
a = 0;                  // OK: これは代入
```

---

while、if、forの直後にセミコロン(;)がある場合。以下はエラーになり、意図せぬ無限ループとなります。

---

```
while (i++);           // WARNING: 予期せぬ無限ループ
```

---

意図的に無限ループを作りたいときは、whileとセミコロンの間にスペースかコメントを挿入してください。例えば、以下はエラーになりません。

---

```
while (i++) ;           // OK: スペースで区切る
while (i++) /* コメントで区切る */ ;
```

---

[ 間違えやすいエラー ] オプションは、プラグマ `warn_possunwant` に相当します ([「warn\\_possunwant」\(p155\)](#))。このプラグマが on か否かをチェックするには、`__option(warn_possunwant)` を使います ([「オプションのチェック」\(p162\)](#))。

## 未使用変数

[ 未使用変数 ] オプションがオンの場合、宣言してあるが使われていない変数を見つけると警告を出します。このオプションは、変数名のミススペルを見つけるためや書き忘れた変数を見つけるのに役立ちます。

---

```
void foo(void)
{
    int temp, error;           // ERROR: error はミススペル
    error = do_something()     // WARNING: temp と error は未使用
}
```

---

使用しない変数を宣言しておく必要があるときは、次の例のようにプラグマ `unused` を使ってください。

---

```
void foo(void)
{
    int i, temp, error;
```

---

```
#pragma unused (i, temp) /* コンパイラは i と temp が未使用
error=do_something();    /* であることを警告しない */

}
```

---

[ 未使用変数 ] オプションはプラグマ `warn_unusedvar` に相当します  
([「warn\\_unusedvar」\(p158\)](#))。このプラグマが on か否かをチェックするには、`__option`  
(`warn_unusedvar`) を使います ([「オプションのチェック」\(p162\)](#))。

## 未使用引数

[ 未使用引数 ] オプションがオンの場合、宣言してあるけれども使われていない引数を見つけると警告を出します。このチェックは、引数名のミススペルを見つけるためや書き忘れた引数を見つけるのに役立ちます。

```
void foo(int temp,int error); // ERROR: error はミススペル
{
    error = do_something(); // WARNING: temp と error は未使用
}
```

---

使用しない引数を宣言しておく必要があるときは、次の例のようにプラグマ `unused` を使ってください。

```
void foo(int temp, int error)
{
    #pragma unused (temp)
    /* コンパイラは temp が未使用であることに警告しない */
    error=do_something();
}
```

---

[ ANSI に厳格に従う ] オプションをオフにして、以下のように使用しない変数名を書かないようにすることもできます ([「関数内で名前のない引数」\(p34\)](#))。

```
void foo(int /* temp */, int error)
{
    /* コンパイラは temp が未使用である、名前がないことを警告しない

    error=do_something(); */
}
```

---

[ 未使用引数 ] オプションはプラグマ `warn_unusedarg` に相当します  
([「warn\\_unusedarg」\(p157\)](#))。このプラグマが on か否かをチェックするには、`__option`  
(`warn_unusedarg`) を使います ([「オプションのチェック」\(p162\)](#))。

## 余分なコンマ

[ 余分なコンマ ] オプションがオンの場合、コンパイラが不要なコンマを見つけると警告を出します。例えば以下のものは C としては正しいですが、このオプションがオンだと警告が表示されます。

---

```
int a[] = { 1, 2, 3, 4, }; // ^ WARNING:4 の後に余分なコンマ
```

---

[ 余分なコンマ ] オプションはプラグマ `warn_extracomma` に相当します (「[warn\\_extracomma](#)」(p152))。このプラグマが on か否かをチェックするには、`__option (warn_extracomma)` を使ってください (「[オプションのチェック](#)」(p162))。

## 拡張エラーチェック

[ 拡張エラーチェック ] オプションがオンの場合、C コンパイラが文法上の問題を見つけると (エラーではなく) 警告を出します。

`void` 宣言されていない関数が `return` を含まない場合。以下は警告になります。

---

```
main()          /* assumed to return int */
{
    printf ("hello world\n");
} /* WARNING: return ステートメントがない */
```

---

以下は問題ありません。

---

```
void main() /* return void を戻す関数 */
{
    printf ("hello world\n");
}
```

---

`enum` 型 ( 列挙型 ) に整数または浮動小数点値を代入した場合。

---

```
enum Day { Sunday, Monday, Tuesday, Wednesday,
            Thursday, Friday, Saturday } d;

d = 5;          /* WARNING */
d = Monday;     /* OK */
d = (Day)3 ;    /* OK */
```

---

`void` 宣言されていない関数に、空の `return` ステートメント (`return;`) がある場合。以下は警告になります。

---

```
int MyInit(void)
{
    int err = GetMyResources();
```

```
if (err !=0) return; /* ERROR: 空の return ステートメント */

/* ... */
```

---

以下は問題ありません。

```
int MyInit(void)
{
    int err = GetMyResources();
    if (err!=0) return -1; /* OK */

    /* ... */
```

---

[ 拡張エラーチェック ] オプションはプラグマ `extended_errorcheck` に相当します ([「`extended\_errorcheck`」\(p96\)](#))。このプラグマが on か否かをチェックするには、`__option (extended_errorcheck)` を使います ([「オプションのチェック」\(p162\)](#))。

## 仮想関数が隠ぺいされた場合

[ 仮想関数が隠ぺいされた場合 ] オプションがオンの場合、スーパークラスの仮想関数を隠す非仮想メンバー関数を宣言すると、コンパイラは警告を出します。ある関数は、同じ名前だけれど異なる引数型をもつ関数を隠します。以下に例を示します。

```
class A {
public:
    virtual void f(int);
    virtual void g(int);
};

class B: public A {
public:
    void f(char);           // WARNING: A::f(int) を隠す
    virtual void g(int);    // OK: A::g(int) をオーバーライド
};
```

---

[ 仮想関数が隠ぺいされた場合 ] オプションはプラグマ `warn_hidevirtual` に相当します ([「`warn\_hidevirtual`」\(p152\)](#))。このプラグマが on か否かをチェックするには、`__option (warn_hidevirtual)` を使います ([「オプションのチェック」\(p162\)](#))。

## 暗黙の算術変換

[ 暗黙の算術変換 ] オプションがオンの場合、計算結果の値が代入先の変数のサイズより大きい場合に警告を出します。例えば、`long` 型変数の値を `char` 型変数に代入した場合、警告を出します。



## 関数がインライン展開されなかった場合

[ 関数がインライン展開されなかった場合 ] オプションがオンの場合、コンパイラは関数をインラインできないときに警告を出します。

このオプションはプラグマ [warn\\_notinlined](#) に相当します。このプラグマが on か否かをチェックするには、`__option (warn_notinlined)` を使います(「[オプションのチェック](#)」(p162))。

## キーワード 'class' と 'struct' の一貫性のない使用

[ キーワード 'class' と 'struct' の一貫性のない使用 ] オプションがオンの場合、コンパイラは `class` と `struct` のキーワードが同じ識別子の定義と宣言で使用されているときに警告を出します。

---

```
class X;  
struct X { int a; }; // warning
```

---

クラスと構造体変数のマングル名に違いを付けるコンパイラで作成されたオブジェクトコード（静的または動的ライブラリ）とリンクするときにはこの警告を使います。

このオプションはプラグマ [warn\\_structclass](#) に相当します。このプラグマが on か否かをチェックするには、`__option (warn_structclass)` を使います(「[オプションのチェック](#)」(p162))。



## 第 3 章 C コンパイラ

この章では、CodeWarrior C コンパイラの C 言語の実装について説明します。

### C コンパイラの概要

この章ではすべての CodeWarrior ターゲットに共通する CodeWarrior C コンパイラについて説明します。この章の情報のほとんどはすべてのオペレーションシステムおよびプロセッサに適用できます。

他の章では、特定のオペレーションシステムまたはプロセッサに特有のコンパイラの機能について説明します。理解を深めるためには、ターゲットに関係のある章をすべて参照してください。

この章では C++ の機能については述べていません。C++ 言語については [「C++ コンパイラの概要」\(p47\)](#) を参照してください。

[「C の実装」\(p27\)](#) : CodeWarrior における C 言語規格の実装について説明します。

[「ANSI/ISO C の拡張」\(p32\)](#) : CodeWarrior C の C 規格の拡張について説明します。

『The C Programming Language, Second Edition』(Kernighan、Ritchie 著、Prentice Hall) の『Appendix A: Reference Manual』へのリファレンスを (K&R、A) と記載しています。これは文中で説明している内容を詳細に解説している ARM のページ番号を示しています。

### C の実装

CodeWarrior が C プログラミング言語の多くの要素をどのように実装しているのかについて説明します。C++ 言語に特有の要素については [「C++ コンパイラの概要」\(p47\)](#) を参照してください。

[識別子](#)

[インクルードファイル](#)

[プリフィックスファイル](#)

[Sizeof\(\) オペレータ](#)

[Volatile 変数](#)

[enum は常に Int 型](#)

## 識別子

(K&R、A2.3)C コンパイラではあらゆるサイズの識別子を作ることができます。しかし、内部 / 外部リンクのときには最初の 255 文字だけが有効です。

## インクルードファイル

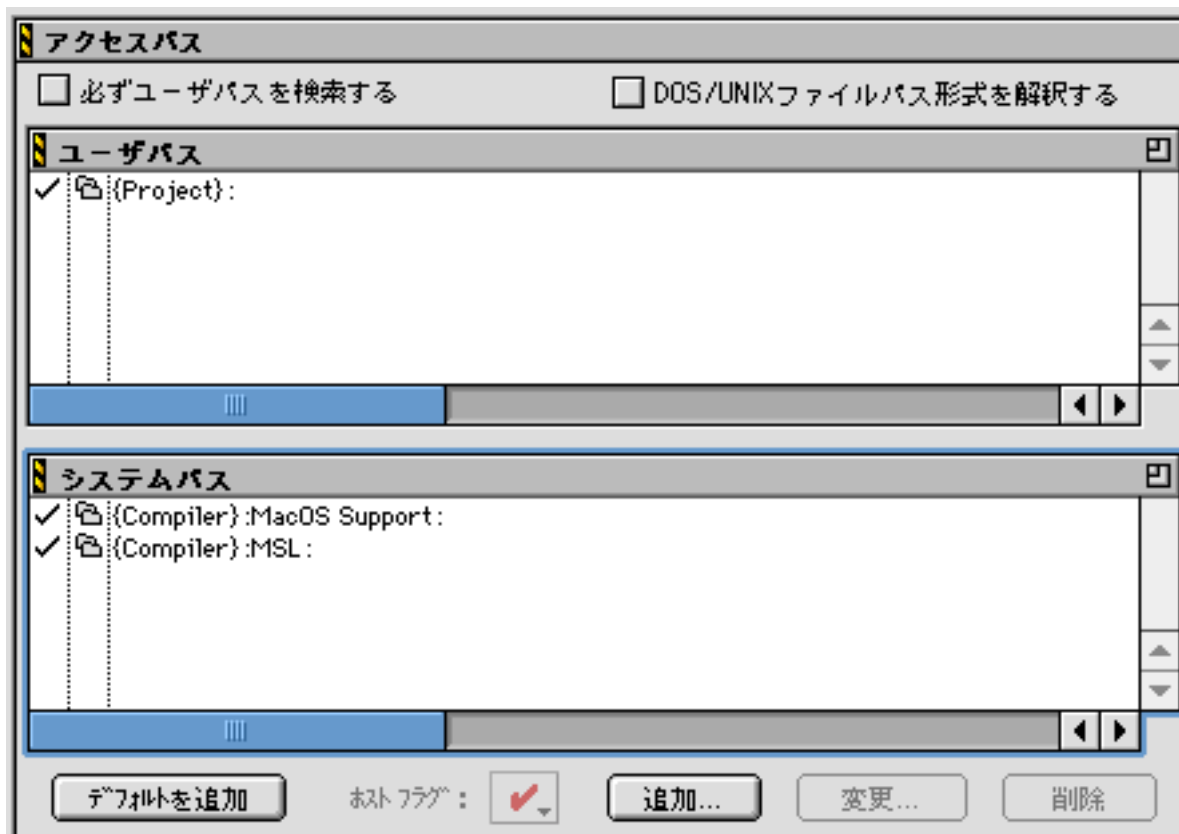
(K&R、A12.4) C コンパイラでは、`#include` ファイルのネストを 32 回まで許可しています。あるインクルードファイルは、他の `#include` ファイル内部の `#include` ステートメントでそのインクルードファイルを使っていれば、ネストされます。例えば、`Main.c` が `MyFunctions.h` をインクルードしていて、それが `MyUtilities.h` を含んでいるならば、`MyUtilities.h` は 1 回ネストされます。

Mac OSでは、次のように`#include`疑似命令にフルパスネームを指定することもできます。

```
#include "HD:Tools:my_headers:macros.h"
```

CodeWarrior IDE では、コンパイラが `#include` ファイルを検索する場所を、アクセスパス設定パネル ( [図 3.1](#) ) で指定します。[ ユーザーパス ] と [ システムパス ] というボタンで 2 つの欄の表示を切り替えます。デフォルトでは、それぞれの欄は 1 つのフォルダを含んでいます。[ ユーザーパス ] 欄は、プロジェクトファイルがある `{Project f}` と、その下のすべてのフォルダを含んでいます。[ システムパス ] 欄は、コンパイラがある `{Compiler f}` とその下のすべてのフォルダを含んでいます。

図 3.1 アクセスパス設定パネル



コンパイラはファイル名を囲む文字によって、`#include` ファイルを [ システムパス ] 欄から、または [ ユーザーパス ] 欄と [ システムパス ] 欄から探し出します。ファイル名をブラケットでくる (例: `#include <stdio.h>`) と、コンパイラは [ システムパス ] 欄のフォルダを検索します。ファイルをダブルクオートでくる (例: `#include "myfuncs.h"`) と、コンパイラは [ ユーザーパス ] 欄のフォルダを検索した後 [ システムパス ] 欄のフォルダを検索します。一般的に、多くのプロジェクトに共通するヘッダファイルにはブラケットを、ある特定のプロジェクト用のヘッダファイルにはダブルクオートを使ってください。

[ 必ずユーザーパスを検索する ] オプションがオンの場合、コンパイラはブラケットで括られたインクルードファイル (`#include <file>`) を [ ユーザーパス ] 欄から探します。

アクセスパス設定パネルにフォルダを追加する方法は『IDE User Guide』を参照してください。

[「プリフィックスファイル」\(p30\)](#) も参照してください。

ヒント： MPW 環境下でコンパイラを使っている場合、『Command-Line Tools Manual』や『MPW Command Reference』にあるように、`#include` ファイルを探しに行く場所を `-i` コンパイラオプションや `{CIncludes}` 変数で指定することができます。

---

## プリフィックスファイル

[C/C++ 言語設定パネル](#)の[プリフィックスファイル]フィールドで、1つのファイルをプロジェクト内のすべてのファイルにインクルードすることができます。[プリフィックスファイル]フィールドにファイルの名前を入力します。

コンパイラはこのファイルを自動的に、プロジェクトのすべてのソースファイルヘインクルードします。プリコンパイルヘッダファイルをプロジェクトにインクルードするには一番便利な方法です。

このフィールドを `-d` オプション(通常コンパイラにコマンドラインの処理を命じる)の代わりに使うこともできます。

参照：[「インクルードファイル」\(p28\)](#)

## Sizeof() オペレータ

`sizeof()` オペレータは、`stddef.h` 内で `unsigned long int` として定義されている `size_t` 型の数を返します。`sizeof()` が `int` 型の数を返すと想定している場合、コードは正常に動作しないかもしれません。

## Volatile 変数

(K&R、A4.4)変数を `volatile` 宣言すると、C コンパイラは以下のように処理します。

変数をレジスタにはストアしない

コードが変数を参照する度に、そのアドレスが再計算される

[例 3.1](#) は、`volatile` 変数の例です。

### 例 3.1 volatile 変数

---

```
void main(void)
{
    int i[100];
    volatile int a, b;

    a = 5;
    b = 20;

    i[a + b] = 15;
    i[a + b] = 30;
}
```

---

コンパイラは、a、b、a+bのいずれもレジスタにはストアしません。同時に、a+bは2行とも別々に再計算されます。

## enum は常に Int 型

(K&R、A8.4) ここではC/C++ が、基礎となる int 型から enum 型をどのように選択するのかを説明します。[C/C++ 言語設定パネル](#)の [ Enum は常に Int 型 ] オプションでコンパイラの挙動を設定します。

[ Enum は常に Int 型 ] オプションの設定によって、2つの処理方法があります。

[ Enum は常に Int 型 ] がオンの場合、ベース型は常に signed int です。enum 定数は signed int よりも小さくなくてはなりません。enum 定数が int 型よりも大きいとき、コンパイラはエラーを発生します。

しかし [ ANSI に厳格に従う ] オプションがオフの場合、unsigned int として表示可能な enum 定数は暗黙に signed int に変換されます。

---

```
#pragma enumsalwaysint on
#pragma ANSI_strict on
enum foo { a=0xFFFFFFFF }; // ERROR. a は 4,294,967,295:
                             // signed int には大きすぎる

#pragma ANSI_strict off
enum bar { b=0xFFFFFFFF }; // OK: b は unsigned int として表示可能
                             // だが、暗黙に signed int (-1) に
                             // 変換される
```

---

その他の機能については「[ANSI に厳格に従う](#)」(p33) を参照してください。

[ Enum は常に Int 型 ] オプションがオフの場合、コンパイラは最大の enum 定数をサポートする int データ型を選択します。このデータ型は最小で char、最大で long になります。64 ビットの long long の値にもなります。

[ Enum は常に Int 型 ] オプションがオフの場合、コンパイラは次のいずれかをピックアップします。

enum 定数がすべて正の場合、コンパイラはすべての enum 定数を代表する大きさを持つ最小の unsigned int のベース型をピックアップします。

最低でも1つの enum 定数が1つでも負の場合、コンパイラはすべての enum 定数を代表する大きさを持つ最小の signed int ベース型をピックアップします。

---

```
#pragma enumsalwaysint off
enum { a=0,b=1 };           // ベース型: unsigned char
enum { c=0,d=-1 };          // ベース型: signed char
enum { e=0,f=128,g=-1 };    // ベース型: signed short
```

---

プラグマ [longlong\\_enums](#) が on の場合のみ、コンパイラは long long 型を使います ( プラグマ longlong\_enums に相当する設定パネルのオプションはありません )

---

```
#pragma enumsalwaysint off
#pragma longlong_enums off
enum { a=0x7FFFFFFFFFFFFFFFFF }; // ERROR: a is too large
#pragma longlong_enums on
enum { b=0x7FFFFFFFFFFFFFFFFF }; // OK: ベース型: signed long long
enum { c=0x8000000000000000 }; // OK: ベース型: unsigned long long
enum { d=-1,e=0x80000000 }; // OK: ベース型: signed long long
```

---

プラグマ longlong\_enums が off、かつ [ ANSI に厳格に従う ] オプションがオンの場合、大きな符号なし 32 ビットの enum 定数( 0x7FFFFFFFF よりも大きいもの )と負の enum 定数を一緒に使うことはできません。プラグマ longlong\_enums<sup>™</sup>off、かつ [ ANSI に厳格に従う ] オプションもオフのとき、大きな符号なし 32 ビットの enum 定数は暗黙に符号付き 32 ビット型に変換されます。

---

```
#pragma enumsalwaysint off
#pragma longlong_enums off
#pragma ANSI_strict on
enum { a=-1,b=0xFFFFFFFF }; // error
#pragma ANSI_strict off
enum { c=-1,d=0xFFFFFFFF }; // ベース型: signed int (b== -1)
```

---

[ Enum は常に Int 型 ] オプションはプラグマ [enumsalwaysint](#) に相当します。このプラグマが on か否かをチェックするには \_\_option (enumsalwaysint) を使います。デフォルトではこのオプションはオフです。

参照: [「enumsalwaysint」\(p93\)](#)、[「longlong\\_enums」\(p111\)](#)、[「オプションのチェック」\(p162\)](#)

## ANSI/ISO C の拡張

ここでは、すべてのターゲットに共通する C 言語の拡張仕様について説明します。ほとんどの拡張は [C/C++ 言語設定パネル](#) のオプションでオンとオフを設定できます。詳しくは [「C/C++ 言語設定パネル」\(p15\)](#) を参照してください。

[「ANSI に厳格に従う」\(p33\)](#)

[「C++ 形式のコメント」\(p34\)](#)

[「関数内で名前のない引数」\(p34\)](#)

[「マクロ定義内で引数を伴わない #」\(p35\)](#)

[「#endif 後の識別子」\(p35\)](#)

[「キャストされたポインタを lvalue として使用」\(p36\)](#)

[「アドレスによる変数宣言」\(p36\)](#)



[「ANSI キーワードのみ」\(p36\)](#) y」  
[「Trigraph 拡張」\(p37\)](#)  
[「整数値としてのキャラクタ定数」\(p37\)](#)  
[「関数のインライン」\(p38\)](#)  
[「マルチバイト文字（日本語など）に対応」\(p39\)](#)  
[「文字列定数を一カ所にまとめる」\(p39\)](#)  
[「文字列定数を再利用しない」\(p40\)](#)  
[「関数プロトタイプが必要」\(p41\)](#)  
[「CR の代わりに NL を利用」\(p42\)](#)  
[「ポインタタイプリールを緩める」\(p43\)](#)  
[「unsigned char を使用」\(p43\)](#)  
[「64 ビット整数を使う」\(p43\)](#)  
[「ポインタを同じサイズの型へ変換する」\(p44\)](#)  
[「コンパイル時にアラインメントと型情報を取得」\(p44\)](#)  
[「構造体内のゼロ長の配列」\(p44\)](#)  
[「ビット回転の組み込み関数」\(p45\)](#)  
[「D 定数接尾子」\(p45\)](#)  
[「short double 型」\(p45\)](#)  
[GNU C 拡張](#)

ターゲットに特有の拡張については各『Targeting』マニュアルを参照してください。

## ANSI に厳格に従う

[C/C++ 言語設定パネル](#)の [ ANSI に厳格に従う ] オプションをオンにすると ANSI の仕様に厳密な仕様となります。

[ ANSI に厳格に従う ] オプションがオフの場合、CodeWarrior コンパイラが提供する以下の C 言語の拡張仕様を設定できます。

[wchar\\_t 型サポート](#)  
[C++ 形式のコメント](#)  
[関数内で名前のない引数](#)  
[マクロ定義内で引数を伴わない #](#)  
[#endif 後の識別子](#)  
[キャストされたポインタを lvalue として使用](#)

### [ポインタを同じサイズの型へ変換する](#)

### [構造体内のゼロ長の配列](#)

### [D 定数接尾子](#)

[ ANSI に厳格に従う ] オプションを個別に設定することはできません。

このオプションは `enum` 定数の処理方法に影響します。[「enum は常に Int 型」\(p31\)](#) を参照してください。

このオプションは C++ プログラムの `main()` 関数に影響します。[「main\(\) の暗黙のリターンステートメント」\(p48\)](#) を参照してください。

[ ANSI に厳格に従う ] オプションはプリグマ [ANSI\\_strict](#) に相当します。このプリグマが `on` か否かをチェックするには `__option (ANSI_strict)` を使います。

参照：[「ANSI\\_strict」\(p77\)](#)、[「オプションのチェック」\(p162\)](#)

## wchar\_t 型サポート

[ `wchar_t` 型サポート ] オプションがオンの場合、ワイドキャラクタを表す標準 C++ の `wchar_t` 型を使うことができます。オフの場合、通常の `char` 型を使います。

## C++ 形式のコメント

( K&R、A2.2 ) C コンパイラでも C++ 形式のコメントが使えます。// に続く行は、コメントとみなされます。

---

```
a = b; // これは C++ のコメント
```

---

この機能を使うには、[C/C++ 言語設定パネル](#)の [ ANSI に厳格に従う ] オプションをオフにします。[ ANSI に厳格に従う ] がオンの場合、この C 規格の拡張は使用できません。

参照：[「ANSI に厳格に従う」\(p33\)](#)

## 関数内で名前のない引数

( K&R、A10.1 ) C コンパイラでは、関数定義内で名前のない引数を使うことができます。

---

```
void f(int ) { } /* OK, [ ANSI に厳格に従う ] がオフの場合 */  
void f(int i) { } /* 常に OK */
```

---

この機能を使うには、[C/C++ 言語設定パネル](#)の [ ANSI に厳格に従う ] オプションをオフにします。[ ANSI に厳格に従う ] がオンの場合、この C 規格の拡張は使用できません。

参照：[「ANSI に厳格に従う」\(p33\)](#)

## マクロ定義内で引数を伴わない #

(K&R, A12.3) C コンパイラは、マクロ定義内で引数を伴わない # を使ってもエラーにしません。

---

```
#define add1(x) #x #1 // OK だが期待通りではない
                        // add1(abc) は "abc"#1 を作成する
#define add2(x) #x "2" // OK: add2(abc) は "abc2" を作成する
```

---

この機能を使うには、[C/C++ 言語設定パネル](#)の [ ANSI に厳格に従う ] オプションをオフにします。[ ANSI に厳格に従う ] がオンの場合、この C 規格の拡張は使用できません。

参照: [「ANSI に厳格に従う」\(p33\)](#)

## #endif 後の識別子

(K&R, A12.5) C コンパイラでは、#endif や #else の後に識別子を置くことが許されます。この拡張により、#endif と、#if、#ifdef、#ifndef との対応関係が以下のようにわかりやすくなります。

---

```
#ifdef __MWERKS__
#   ifndef __cplusplus
        /*
         * . . .
         */
#   endif __cplusplus
#endif __MWERKS__
```

---

この機能を使うには、[C/C++ 言語設定パネル](#)の [ ANSI に厳格に従う ] オプションをオフにします。[ ANSI に厳格に従う ] がオンの場合、この C 規格の拡張は使用できません。

参照: [「ANSI に厳格に従う」\(p33\)](#)

---

ヒント: [ ANSI に厳格に従う ] オプションをオンにした (拡張を無効にしたとき) 場合、以下のように識別子をコメントにすることもできます。

---

---

```
#ifdef __MWERKS__
#   ifndef __cplusplus
        /*
         * . . .
         */
#   endif /* __cplusplus */
#endif /* __MWERKS__ */
```

---

## キャストされたポインタを lvalue として使用

C コンパイラでは、キャストされたポインタを lvalue として使用することができます。

---

```
char *cp;  
((long *) cp)++; /* [ANSI に厳格に従う] がオフなら OK */
```

---

この機能を使うには、[C/C++ 言語設定パネル](#)の [ANSI に厳格に従う] オプションをオフにします。[ANSI に厳格に従う] がオンの場合、この C 規格の拡張は使用できません。

参照：[「ANSI に厳格に従う」\(p33\)](#)

## アドレスによる変数宣言

(K&R、A8.7) C コンパイラでは、変数が参照するアドレスを指定することができます。次の例では MemErr を 0x0220 番地のアドレスにある変数と定義します。

---

```
short MemErr:0x220;
```

---

変数 MemErr は、0x0220 番地のアドレスにあります。

---

**警告!** Mac OS でのプログラミングでは、ローメモリグローバルを参照するときにこの拡張を使うのは避けてください。プログラムが MacOS の将来のバージョンとも互換性があることを保証するために、LowMem.h ヘッダファイルで定義された関数を使ってください。

---

この拡張はオフにできません。これに相当するプリAGMAやオプションはありません。

## ANSI キーワードのみ

(K&R、A2.4) C コンパイラが認識できる追加キーワードがいくつかあります。[C/C++ 言語設定パネル](#)の [ANSI キーワードのみ] オプションでそれらを認識するか否かを設定します。

[ANSI キーワードのみ] がオンの場合、コンパイラが Metrowerks C の追加キーワードを見つけるとエラーを発生します。ANSI/ISO 規格に厳密に従うコードを書く場合、[ANSI キーワードのみ] オプションをオンにしてください。

[ANSI キーワードのみ] がオフのとき、以下の追加キーワードが使用可能になります。

asm：関数の内容をビルトインのインラインアセンブラでコンパイルします (K&R、A10.1)

far：Motorola 68K プロセッサ用の CodeWarrior C/C++ コンパイラは far をキーワードとして扱います。それ以外の CodeWarrior C/C++ コンパイラは far をキーワードとして認識しません。

`inline` : C 関数がインラインであることを宣言します。詳細は「[関数のインライン](#)」(p38)を参照してください。

`pascal` : Mac OS プログラミングで使われるキーワードです。

`__stdcall` : Win32 プログラミングで使われるキーワードです。

`vector, pixel` : PowerPC Altivec オブジェクトコードを生成するキーワードです。  
「[altivec\\_codegen](#)」(p75)、「[altivec\\_model](#)」(p76)、「[altivec\\_codegen](#)」(p75)を参照してください。

[ ANSI キーワードのみ ] オプションはプラグマ [only\\_std\\_keywords](#) に相当します。このプラグマが on か否かをチェックするには、`__option (only_std_keywords)` を使います。デフォルトでは、このオプションはオフです。

参照 : 「[only\\_std\\_keywords](#)」(p119)、「[オプションのチェック](#)」(p162)

Trigraph 拡張

(K&R、A12.1) C コンパイラは、Trigraph 文字を無視できます。一般的な文字定数 (特に Mac OS のもの) の多くは Trigraph 文字列に似ているので、この拡張によってそれらをエスケープキャラクタなしで使うことができます。

ANSI/ISO 規格に厳密に従わなければならないコードを書く場合、[C/C++ 言語設定パネル](#)の [ Trigraph 拡張 ] オプションをオンにしてください。? マークを含む文字列やマルチキャラクタ定数を初期化するときに注意が必要です。

```
char c = '????'; // ERROR: Trigraph シーケンスは '??' へ展開される。
char d = '\?\?\?\?'; // OK
```

[ Trigraph 拡張 ] オプションはプラグマ [trigraphs](#) に相当します。このプラグマが on か否かをチェックするには、`__option (trigraphs)` を使います。デフォルトでは、このオプションはオフです。

参照 : 「[trigraphs](#)」(p148)、「[オプションのチェック](#)」(p162)

整数値としてのキャラクタ定数

(K&R、A2.5.2)C コンパイラでは 2 ~ 4 文字のキャラクタ文字列定数を 32 ビット整数値として使うことができます。

表 3.1      キャラクタ文字列定数と整数値

文字定数	等価の 16 進法
'ABCD'	0x41424344
'ABC'	0x00414243
'AB'	0x00004142

この拡張はオフにできません。これに相当するプリAGMAやオプションは [C/C++ 言語設定パネル](#) にありません。

注意： この機能とマルチバイト文字列は別のものです。マルチバイト文字列は 1 バイトよりも大きいデータ型で表されます。

256 文字以上の文字セット（漢字など）を使うための拡張については「[マルチバイト文字（日本語など）に対応](#)」(p39) を参照してください。

関数のインライン

コンパイラは関数をインライン化するか否かを、[C/C++ 言語設定パネル](#)の「[ANSI キーワードのみ](#)」オプションと「インラインの深さ」ポップアップメニュー、「自動インライン展開」オプション、「定義前のインライン展開を許可」オプションの設定で判断します。

初心者： インライン関数を呼ぶとき、コンパイラは関数を呼ぶ代わりにその関数コードを挿入します。関数をインライン化することで、プログラムが高速化されます（コンパイラはその関数を呼び出さずに、コードをすぐに実行するからです）が、プログラムサイズが大きくなります（関数コードがいくつかの異なる箇所でも繰り返して使われるためです）。

「ANSI キーワードのみ」オプションがオフの場合、C 関数を inline 宣言することができます。[表 3.2](#) に示すように [C/C++ 言語設定パネル](#)の「インラインの深さ」ポップアップメニューでインライン化する関数のレベルを選択できます。

表 3.2 「インラインの深さ」ポップアップメニュー

選択肢	内容
インライン化しない	全くインライン化しません。C/C++ 関数が inline 宣言されていても行いません。
スマート	2 ~ 4 の深度の関数をインライン化します。
1 ~ 8	選択した数値の深度の関数をインライン化します。

「インラインの深さ」ポップアップメニューの「スマート」と、「1 ~ 8」はプリAGMA inline\_depth に相当します（参照：「[C/C++ 言語設定パネル](#)」(p15)）。このプリAGMAが on か否かをチェックするには、\_\_option (inline\_depth) を使います（参照：「[オプションのチェック](#)」(p162)）。

「インライン化しない」はプリAGMA dont\_inline に相当します（「[dont inline](#)」(p92)）。このプリAGMAが on か否かをチェックするには、\_\_option (dont\_inline) を使います（「[dont inline](#)」(p164)）。デフォルトではこのオプションはオフです。

「自動インライン展開」オプションによって、コンパイラはインライン化する関数を選択します。また inline 宣言された C++ 関数とクラス宣言で定義されたメンバー関数もインライ

ン化します。このオプションはプラグマ `auto_inline` に相当します([「auto\\_inline」\(p79\)](#))。このプラグマが on か否かをチェックするには、`__option (auto_inline)` を使います([「auto\\_inline」\(p163\)](#))。デフォルトではこのオプションはオフです。

[ 定義前のインライン展開を許可 ] オプションがオンの場合、コンパイラはまだ定義されていない関数をインライン化します。このオプションはプラグマ `auto_inline` に相当します([「defer\\_codegen」\(p85\)](#))。このプラグマが on か否かをチェックするには、`__option (defer_codegen)` を使います([「defer\\_codegen」\(p163\)](#))。

[ 定義前のインライン展開を許可 ] オプションがオンの場合、コンパイラは他の関数から参照される関数を、定義よりも先にインライン化するためにより多くのメモリをコンパイル時に使います。オフの場合、コンパイラは定義よりも先に参照される関数をインライン化しません。たとえ関数が明示的にインライン定義されている場合でもインライン化しません。

## マルチバイト文字（日本語など）に対応

C コンパイラは、2 バイト以上の文字を使う言語（Unicode や漢字など）もサポートします。この機能は [C/C++ 言語設定パネル](#) の [ マルチバイト文字（日本語など）に対応 ] オプションで設定します。

マルチバイト文字列やコメントを使用するには、[ マルチバイト文字（日本語など）に対応 ] オプションをオンにしてください。マルチバイト文字列やコメントを使わないのなら、コンパイラのスピードを落とさないために、このオプションをオフにしてください。

複数キャラクタで構成されるキャラクタ定数を作成する方法については [「整数値としてのキャラクタ定数」\(p37\)](#) を参照してください（これは名前は似ていますが全く違う問題です）。

## 文字列定数を一カ所にまとめる

C/C++ 言語設定パネルの [ 文字列定数を一カ所にまとめる ] オプションは、コンパイラがどのように文字列定数を保存するかを設定します。

---

注意： 原則的にこのオプションはすべてのターゲットに使用できます。しかし、これが有効なのは PowerPC プロセッサや Code Fragment Manager と 68K プロセッサを搭載した Mac OS で使用されている Table of Contents ベース（TOC ベース）のリンクメカニズムに対してのみです。

---

[ 文字列定数を一カ所にまとめる ] オプションがオンの場合、コンパイラはすべての文字列定数を 1 つのデータオブジェクトに入れるので、それらすべてに対して 1 つの TOC エントリだけが必要になります。このオプションをオンにすると、プログラム中の TOC エントリ数は減りますが、プログラムサイズは増えます。文字列のアドレスを保存するのは効率的な方法ではないためです。

[ 文字列定数を一カ所にまとめる ] がオフの場合、コンパイラは文字列定数に対して、個々のデータオブジェクトとそれぞれの TOC エントリを作ります。



---

ヒント： TOC のサイズは、PPC プロセッサ設定パネルの [ 小さな Static データを TOC 内に保存する ] オプションで変更できます。詳細は『Targeting Mac OS』マニュアルを参照してください。

---

このオプションは、プログラムが大きくて多くの文字列定数がある場合に特に有効です。

---

注意： [ 文字列定数を一ヶ所にまとめる ] オプションをオンにすると、コンパイラは 68K プロセッサ設定パネルの [ PC- 相対文字列 ] オプションを無視します。これは 68K 専用の機能です。

---

[ 文字列定数を一ヶ所にまとめる ] オプションはプラグマ [pool\\_strings](#) に相当します。このプラグマが on か否かをチェックするには、`__option (pool_strings)` を使います。デフォルトではこのオプションはオフです。

参照：[「pool\\_strings」\(p129\)](#)、[「オプションのチェック」\(p162\)](#)

## 文字列定数を再利用しない

[C/C++ 言語設定パネル](#)の [ 文字列定数を再利用しない ] オプションは、コンパイラがどのように文字列リテラルを保存するかを設定します。

[ 文字列定数を再利用しない ] がオンの場合、コンパイラはそれぞれの文字列を別々に保存します。

[ 文字列定数を再利用しない ] がオフの場合、コンパイラは同一の文字列を 1 つだけ保存します。このオプションは変更しない同一の文字列がたくさんあるときにメモリを節約するのに役立ちます。

オフの場合（つまり文字列が同一の文字列に再使用された場合）に文字列を 1 つでも変更すると、すべての文字列定数が変更されます。

---

```
char *str1="Hello";  
char *str2="Hello"; // 2 つの同じ文字列  
*str2 = 'Y';
```

---

[ 文字列定数を再利用しない ] がオンの場合、文字列は別々に保存されます。1 文字目を変更すると、`str1` が "Hello" のままで、`str2` が "Yello" になります。

[ 文字列定数を再利用しない ] がオフの場合（つまり同じメモリロケーションが再使用されている場合）、2 つの文字列は全く同じであるため、同じメモリロケーションに保存されます。1 文字目を変更すると、`str1`、`str2` 共に "Yello" になります。これによってバグの発見が困難になります。

[ 文字列定数を再利用しない ] オプションはプラグマ [dont\\_reuse\\_strings](#) に相当します。このプラグマが on か否かをチェックするには、`__option (dont_reuse_strings)` を使います。デフォルトではこのオプションはオンです（文字列は再使用されません）。



参照: [「dont\\_reuse\\_strings」\(p92\)](#)、[「オプションのチェック」\(p162\)](#)

## 関数プロトタイプが必要

(K&R、A8.6.3、A10.1) C コンパイラでは、関数のプロトタイプの強制を選択できます。C/C++ 言語設定パネルの[ マルチバイト文字(日本語など)に対応 ]オプションで設定します。

[ 関数プロトタイプが必要 ] オプションがオンの場合、参照語に宣言されていて、かつプロトタイプ宣言のない関数を使うとコンパイラがエラーを出します。関数が参照される前に宣言されていて、かつプロトタイプがない場合、コンパイラは警告を出します。

このオプションは、宣言や定義する前に関数を呼び出すエラーを防ぐのに有効です。例えば、関数プロトタイプがないと、間違った型のデータを渡すことがあります。結果としてコンパイラがエラーを出さなくても、コードが期待通りに動作しないことがあります。

[例 3.2](#) では、`PrintNum()` は整数引数と呼ばれますが、後に浮動小数点引数として定義されています。

### 例 3.2 目立たないタイプミスマッチ

---

```
#include <stdio.h>

void main(void)
{
    PrintNum(1);    // PrintNum() は integer を float として
                   // 変換しようとする。0.000000 と印字
}

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

---

これを実行すると、以下の結果が得られます。

0.000000

コンパイラはタイプミスマッチについてエラーを告げませんが、関数は期待通りに動きません。`PrintNum()` はプロトタイプ宣言されていないので、コンパイラはその関数を呼ぶ前に整数型を浮動小数点型に変換しなければいけないことがわかりません。そして、その関数は受け取ったビットを浮動小数点型の数として扱うので、意味のない結果をプリントします。

[例 3.3](#) のように、まず `PrintNum()` をプロトタイプ宣言しておけば、コンパイラはその引数を浮動小数点型の数に変換し、期待通りの結果がプリントされます。

例 3.3 プロトタイプを使い、タイプミスマッチを避ける

---

```
#include <stdio.h>

void PrintNum(float x); // 関数のプロトタイプ

void main(void)
{
    PrintNum(1);        // コンパイラは int を float へ変換
}                       // 1.000000 と印字

void PrintNum(float x)
{
    printf("%f\n", x);
}
```

---

上記の場合、コンパイラは渡された値を自動的にタイプキャストします。自動型変換が使えないような状況では、関数のプロトタイプが要求するデータ型と引数がマッチしないためコンパイラはエラーを出します。そのような型のミスマッチのエラーはコンパイル時に簡単に見つかります。プロトタイプを使わなければエラーは発生しませんが、予期しない動作になるため解析するのがとても困難になります。

[ Require Prototypes ] オプションはプラグマ [require\\_prototypes](#) に相当します。このプラグマが on か否かをチェックするには、`__option (require_prototypes)` を使います。デフォルトではこのオプションはオンです。

参照 : [「require\\_prototypes」\(p133\)](#)、[「オプションのチェック」\(p162\)](#)

## CR の代わりに NL を利用

C コンパイラでは改行 (`\n`) とリターン (`\r`) の文字の扱いを選択できます。C/C++ 言語設定パネルの [ CR の代わりに NL を利用 ] オプションでこれを設定します。

Metrowerks C/C++ を含めほとんどのコンパイラでは、`\r` はキャリッジリターン (`0x0D`)、`\n` はラインフィード (`0x0A`) に変換されます。

[ Map Newlines to CR ] がオンの場合、Metrowerks C は MPW 規約に準拠して `\n` と `\r` の文字を扱います。

[ CR の代わりに NL を利用 ] がオフの場合、コンパイラはこれらの文字を Metrowerks C/C++ 規約で扱います。

このオプションをオンにするときは、ANSI/ISO C/C++ ライブラリがこのオプションをオンにしてコンパイルされたものかどうかを確認してください。このオプションがオンの場合、標準の [ CR の代わりに NL を利用 ] がオフでコンパイルされた ) ANSI C/C++ ライブラリを使うと、`\n` や `\r` を正しく読み書きできません。例えば、`\n` を印字すると新しい行になる代わりに同じ行の先頭になってしまいます。

このオプションはプラグマ [mpwc\\_newline](#) に相当します。このプラグマが on か否かをチェックするには、`__option (mpwc_newline)` を使います。デフォルトはオフです。

参照：[「mpwc\\_newline」\(p116\)](#)、[「オプションのチェック」\(p162\)](#)

Mac OS プログラミングにおける MPW との互換性について詳しくは『Targeting Mac OS』マニュアルを参照してください。

## ポインタタイプルールを緩める

C/C++ 言語設定パネルの [ ポインタタイプルールを緩める ] オプションがオンの場合、コンパイラは `char*`、`unsigned char*` を同じ型として扱います。ポインタ型の互換性のためにプロトタイプがチェックされているときに、直接ポインタを割り当てることができません。

このオプションは、ANSI/ISO C 規格以前に書かれたコードを使うときに特に有効です。古いコードでは、これらの型を相互変換に使っています。

このオプションは C++ コードには効果はありません。このオプションがオンの場合でも、コンパイラは `char*` と `unsigned char*` を別の型として扱います。

[ ポインタタイプルールを緩める ] オプションはプラグマ [mpwc\\_relax](#) に相当します。このプラグマが on か否かをチェックするには、`__option (mpwc_relax)` を使います。

参照：[「mpwc\\_relax」\(p117\)](#)、[「オプションのチェック」\(p162\)](#)

## unsigned char を使用

C コンパイラは `char` 宣言を自動的に `unsigned char` 宣言として扱うことができます。C/C++ 言語設定パネルの [ Unsigned Char を使用 ] オプションで設定します。

[ Unsigned Char を使用 ] がオンの場合、C コンパイラは `char` 宣言を `unsigned char` 宣言として扱います。

---

注意： このオプションがオンの場合、このオプションがオフでコンパイルされたライブラリとの互換性がなくなることがあります。

---

[ Unsigned Char を使用 ] オプションはプラグマ [unsigned\\_char](#) に相当します。このプラグマが on か否かをチェックするには、`__option (unsigned_char)` を使います。デフォルトではこのオプションはオフです。

参照：[「unsigned\\_char」\(p149\)](#)、[「オプションのチェック」\(p162\)](#)

## 64 ビット整数を使う

C コンパイラでは、`long long` 規則子を使って 64 ビット整数を定義することができます。これはプラグマ [longlong](#) で設定します。C/C++ 言語設定パネルにはこれを設定するオプションはありません。

このプラグマが `on` の場合、`long long` 整数を宣言することができます。`long long` は  $-9,223,372,036,854,775,808$  から  $9,223,372,036,854,775,807$  までの値を保持できます。`unsigned long long` は 0 から  $18,446,744,073,709,551,615$  までの値を保持できます。

このプラグマが `off` のときに `long long` を使うとシンタックスエラーが発生します。

`enum` 型では、`long long` を保持できる大きさの `enum` 定数を使うことができます。詳しくは「[enum は常に Int 型](#)」(p31) を参照してください。`long long` ビットフィールドはサポートされていません。

プラグマ `longlong` で `long long` 型をコントロールできます。このプラグマが `on` か否かをチェックするには、`__option (longlong)` を使います。デフォルトではこのプラグマは `on` です。

参照：「[longlong](#)」(p111)、[「オプションのチェック」](#)(p162)

## ポインタを同じサイズの型へ変換する

C コンパイラはグローバル初期化でポインタ型を同じサイズの整数型へ変換することができます。型変換は ANSI C 規格に準拠していませんので、C/C++ 言語設定パネルの「ANSI に厳格に従う」オプションがオフのときのみ利用できます。詳細は「[ANSI に厳格に従う](#)」(p33) を参照してください。

### 例 3.4 ポインタを同じサイズの整数型へ変換する

---

```
char c;  
long arr = (long)&c; // 受け入れられる (ANSI/ISO C ではない)
```

---

## コンパイル時にアラインメントと型情報を取得

C コンパイラはデータ型とそのバイトアラインメントの情報を戻すビルトイン関数を 2 つ持っています。

関数呼び出し `__builtin_align(typeID)` はデータ型 `typeID` のバイトアラインメントを戻します。

関数呼び出し `__builtin_type(typeID)` はデータ型 `typeID` の種類を示す整数値を戻します。`__builtin_type(typeID)` は、`typeID` が整数あるいは列挙型の場合は 0 を戻します。`typeID` が浮動小数点型であれば、1 を戻します。`typeID` が他のデータ型であれば、2 を戻します。

## 構造体内のゼロ長の配列

[C/C++ 言語設定パネル](#)の「ANSI に厳格に従う」オプションがオフの場合、コンパイラは長さのない配列を構造体の最後のアイテムとして許可します。[例 3.5](#) の例では、ブラケット内のインデックス値をゼロとして、またはインデックス値なしで配列を定義することができます。

## 例 3.5 ゼロ長の配列を使う

---

```
struct listOfLongs {  
    long listCount;  
    long list[0]; // [ANSI に厳格に従う] がオフなら [] も OK  
}
```

---

## ビット回転の組込み関数

CodeWarrior C には以下の、右または左にビットを回転させる関数があります。

```
__rol(op, n)  
__ror(op, n)
```

引数 *op* は、ビットを回転させるアイテムを表します。引数 *n* は *op* ビットで回転させるアイテムの数を表します。引数 *op* は大きなデータ型にはならず、char、short、int、long、long long のいずれかになります。

これらの関数は組込み（ビルトイン）です。つまり、この関数を使うためのプロトタイプは不要で、特殊なライブラリへのリンクも不要です。

---

注意： 現在、これらの関数を使えるのは Motorola 68K および Intel x86 用の CodeWarrior C/C++ コンパイラだけです。

---

## D 定数接尾子

浮動小数点定数値の直後に D があるとき、CodeWarrior コンパイラがはその値を double 型データとして処理します。

プラグマ [float\\_constants](#) が on の場合、浮動小数点定数が D で終わらなければ、それらは float 型の値として扱われます。

参照：「[float\\_constants](#)」(p100)

## short double 型

CodeWarrior C では short double 型を使用できます。ANSI/ISO C 規格はこのデータ型をサポートしていません。Mac OS 用 CodeWarrior C コンパイラは、Mac OS プログラミングで利用されている独特な浮動小数点フォーマットを提供するために、このデータ型を認識します。詳細は『Targeting Mac OS』マニュアルを参照してください。

## GNU C 拡張

プラグマ [gcc\\_extensions](#) が on の場合、CodeWarrior C は非標準の GNU C 言語拡張を受け付けます。

現在、GNU C 言語拡張が認識されるのは、ローカル配列と非定数値の `structs` の初期化です。[例 3.6](#) を参照してください。

例 3.6 配列と構造体を初期化する GNU C 拡張

---

```
void myFunc( int i, double x )
{
    int arr[2] = { i, i + 1 };
    struct myStruct = { i, x };
}
```

---

## 第 4 章 C++ コンパイラ

この章では CodeWarrior C++ コンパイラがどのように C++ 言語を実装しているのかについて説明します。

### C++ コンパイラの概要

ここではすべての CodeWarrior ターゲットに共通する CodeWarrior C++ コンパイラについて説明します。この章の情報のほとんどはすべてのオペレーションシステムおよびプロセッサに適用できます。

他の章では、特定のオペレーションシステムまたはプロセッサに特有のコンパイラの機能について説明します。理解を深めるためにターゲットに関係のある章をすべて参照してください。

C コンパイラは C++ コンパイラの完全な一部分ですので、C コンパイラの機能はすべて C++ コンパイラでも使用できます。この章では C++ コンパイラの機能を説明し、C コンパイラについては述べていません。C++ 言語については [「C コンパイラの概要」\(p27\)](#) を参照してください。

C++ 言語をサポートするコンパイラの機能すべてを説明します。またそれらの設定方法の他にも、RTTI、例外、テンプレートなどの機能についても説明します。

[「C++ の実装」\(p48\)](#) : C++ 規格の一部を Metrowerks C++ がどのように実装しているかについて説明します。

[「サポートされていない拡張」\(p53\)](#) : Metrowerks C++ が現在サポートしていない ANSI/ISO C++ 規格の一般的拡張について説明します。

[「C++ コンパイラの設定」\(p53\)](#) : [C/C++ 言語設定パネル](#) のオプションで Metrowerks C++ の挙動を変更する方法について説明します。

[「C++ 例外処理の使用」\(p57\)](#) : try と catch ステートメントを使って例外処理を行う方法を説明します。

[「RTTI の使用」\(p57\)](#) : ランタイムタイプ情報の利用法を説明します。

[「テンプレートの使用」\(p60\)](#) : テンプレートを定義し宣言するファイルの最良な設定方法について説明します。さらに、テンプレートを明確にインスタンス化するための C++ 規格の追加についても説明します。

Embedded C++ (EC++) については [「C++ とエンベデッドシステムの概要」\(p65\)](#) を参照してください。

この章には『The Annotated C++ Reference Manual』( Ellis & Stroustrup 著、Addison-Wesley ) へのリファレンスを、( ARM ) として記載しています。これは文中で説明されている内容を詳細に説明している ARM の章番号を示しています。

## C++ の実装

ここでは『The Annotated C++ Reference Manual』( Ellis & Stroustrup 著、Addison-Wesley ) で述べられている C++ 規格の一部を Metrowerks C++ がどのように実装しているのかについて説明します。以下の内容について説明します。

[main\(\) の暗黙のリターンステートメント](#)

[キーワード順](#)

[追加キーワード](#)

[条件オペレータの規約](#)

[メンバー関数のデフォルト引数](#)

[インライン関数のローカルクラス宣言](#)

[クラスオブジェクトのコピーと構築](#)

[静的データの初期化用リソースチェック](#)

[継承されたメンバー関数の呼び出し](#)

### main() の暗黙のリターンステートメント

main() 関数が int 型の結果を返し、かつユーザーのリターンステートメントで終わらない場合、コンパイラはプログラムの main() 関数に return 0; という C++ のステートメントを追加します。以下に例を示します。

```
int main() { } // 同等: int main() { return 0; }
main() { }     // 同等: int main() { return 0; }
```

C/C++ 言語設定パネルの [ ANSI に厳格に従う ] オプションがオンの場合、コンパイラは外部の int main() 関数にも追加します。

### キーワード順

( ARM、7.1.2、11.4 ) 宣言文で virtual または friend キーワードを使うとき、それらは宣言の最初のワードでなければなりません。

#### 例 4.1 virtual、friend キーワードの使用

---

```
class foo {
    virtual int f0(); // OK
    int virtual f1(); // ERROR
    friend int f2();  // OK
```



```
int friend f3();    // ERROR
}
```

追加キーワード

( ARM 、 2.4、ANSI 、 2.8 )Metrowerks C++ では、ARM2.4 のシンボルと、ANSI C++ 規格の 2.8 の以下のシンボルをキーワードとして予約しています。

bool	const_cast	dynamic_cast
explicit	false	mutable
namespace	reinterpret_cast	static_cast
true	typeid	using

条件オペレータの規約

( ARM 、 5.16 )コンパイラは条件オペレータの 2 番目と 3 番目の表記をリファレンス変換しません。言い替えれば、2 番目と 3 番目の表記が数値型でなければ、それらは同じ型でなければいけません。

例 4.2 条件オペレータ内の変換

```
class base { };
class derived : public base { };

static void foo(derived i)
{
    base      &a = i;
    derived   &b = i, c;
    c = (sizeof(0) ? a:b); // ERROR: b は (base &) へ変換されていない
    c = (sizeof(0) ? a:(base &)b)    // OK, タイプキャスト
}
```

メンバー関数のデフォルト引数

( ARM 、 8.2.6 )コンパイラは、メンバー関数のデフォルト引数をクラス宣言の最後ではバインド（結びつけ）しません。デフォルト引数が現われる前に、デフォルト引数表記で使われる値を宣言しておかなければいけません。

例 4.3 メンバー関数でのデフォルト引数の使用

```
class foo {
    enum A { AA };
    int f(A a = AA); // OK
    int f(B b = BB); // ERROR: BB は宣言されていない
```

```
enum B { BB };  
};
```

---

## インライン関数のローカルクラス宣言

(ARM、9.8)関数内でローカルクラスを宣言するとき、そのクラスのインライン関数は関数外のローカルタイプや変数にアクセスできません。言い替えれば、コンパイラはそのクラスのインライン関数をグローバルスコープレベルに挿入します。

### 例 4.4 インライン関数のローカルクラス宣言の使用

---

```
int x;  
  
void foo()  
{  
    static int s;  
  
    class local {  
        int f1() { return s; } // ERROR: 's' にアクセスできない  
        int f2() { return local::f1(); } // ERROR: local に  
                                         // アクセスできない  
        int f3() { return x; } // OK  
    };  
}
```

---

## クラスオブジェクトのコピーと構築

(ARM、12.1、12.8) コンパイラは、単純なクラスに対してはコピーコンストラクタやデフォルトの `operator=` を作りません。単純なクラスを以下にまとめます。

基底クラスもしくは単純なクラスだけから派生される

クラスメンバーがないか、単純なクラスメンバーだけを持っている

仮想メンバー関数を持たない

仮想基底クラスを持たない

コンストラクタ関数とデストラクタ関数を持たない

### 例 4.5 コンストラクタ関数

---

```
class Simple { int f; };  
  
void simpleFunc (Simple s1)  
{  
    Simple s2 = Simple(s1); // ERROR: 明示的コピーコンストラクタ呼び出し  
                           // コンパイラはデフォルトコピーコンストラクタを
```

```
        // 生成しない
    Simple s3 = s1; // OK: コンパイラはビットワイズコピーを実行
}
```

コンパイラは、生成された代入あるいはコピーコンストラクタが、仮想基底クラスを表わすオブジェクトを一度だけ割り当て初期化することを保証しません。

## 静的データの初期化用リソースチェック

浮動小数点ユニット (FPU) など、特定のリソースを必要とする静的 C++ オブジェクトを作ることがしばしばあります。`\_\_PreInit\_\_()` という関数 (これは、コンパイラが静的データを初期化する前に呼ばれます) を作ると、それらのリソースをチェックすることができます。リソースのチェックは、`main()` 関数内ではできません。コンパイラは `main()` を呼ぶ前に静的データを初期化するためです。

`\_\_PreInit\_\_()` 関数は次のように宣言してください。

```
extern "C" void __PreInit__(void);
```

注意: PPC コンパイラはこの関数をサポートしていません。

このルーチンは浮動小数点ユニットをチェックします。この場合、`HasFPU()` と `DisplayNoFPU()` の関数は自分で定義する必要があります。

### 例 4.6 静的データ初期化前に FPU をチェック

```
#include <Types.h>
#include <stdlib.h>

extern "C" void __PreInit__(void);

void __PreInit__(void)
{
    if(!HasFPU()) {
        DisplayNoFPU(); // "No FPU" 警告を表示
        abort();        // プログラム実行をアボート
    }
}
```

## 継承されたメンバー関数の呼び出し

(ARM、10.2) ローカル関数をオーバーライドする以外に、継承された仮想メンバー関数を呼び出す方法が 2 つあります。第 1 に、基底クラスか親クラスで定義されたメンバー関数

を参照する方法です。第 2 の方法は便利ですが、他のコンパイラを使う場合はお奨めできません。

継承されたメンバー関数の標準的呼び出し

この方法は ANSI/ISO C++ 規格でサポートされており、メンバー関数をその基底クラスで指定します。

`MyFunc()` 関数を実装した `MyBaseClass` と `MySubClass` の 2 つのクラスがあると想定します。その場合、以下のように基底クラスの `MyFunc()` を呼び出すことができます。

---

```
MyBaseClass::MyFunc();
```

---

しかし、クラス階層を変更したときにこのコードは壊れてしまいます。中間のクラスを追加し、その階層が `MyBaseClass`、`MyMiddleClass`、`MySubClass` となっていると仮定します。そのそれぞれが `MyFunc()` を持っているとします。その場合、このコードは `MyMiddleClass` に追加される挙動をバイパスして `MyBaseClass` の `MyFunc()` を呼び出してしまいます。この方法は期待通りではないかもしれません。この種のコードは予期せぬ結果を招き、結果として発見しにくいバグになるかもしれません。

`inherited` キーワードで継承したメンバー関数を呼び出す

---

注意： `inherited` キーワードは ANSI/ISO C++ 規格ではサポートされていません。  
`CodeWarrior C++` では 1 回の継承だけをサポートします。

---

以下の方法で継承した `MyFunc()` を呼び出すことができます。

---

```
inherited::MyFunc();
```

---

この違いは、コンパイラはコンパイル時に基底クラスを識別することです。このコードは両方の直接の基底クラスを呼び出します。一方は `MyBaseClass` の基底クラス、一方は `MyMiddleClass` の直接の基底クラスです。

`inherited` キーワードの利点とは、後にクラス階層を変更してサブクラスが異なる基底クラスから継承されたとしても、直接の基底クラスはその変更に関係なく呼び出されるということです。

---

```
inherited::func-name(param-list);
```

---

これは、そのクラスの直接の基底クラスの `func-name` を呼び出します。もしもクラスが多重継承のために複数の直接の基底クラスを持ち、コンパイラがどの `func-name` を呼べばよいかわからない場合は、エラーになります。

次の例では、`Q` クラスに挙動を追加してオブジェクトを描く `Q` クラスを作成します。

例 4.7 継承されたメンバー関数の呼び出しに `inherited` キーワードを使用

---

```
class O { virtual void draw(Point); }
class Q : O { void draw(Point); }

void O::draw (Point p)
{
    Rect r = { p.x-5, p.y-5, p.x+5, p.y+5 };
    FrameOval(r);          // Draw an O.
}

void Q::draw (Point p)
{
    inherited::draw(p);    // 基底クラスの挙動を実行
    MoveTo(p.x, p.y);      // 追加の挙動を実行
    Line(5, 5);
}
```

---

`inherited` キーワードを使う前に、この疑似命令を挿入してください。

```
#pragma def_inherited on
```

このプラグマの詳細は「[def\\_inherited](#)」(p85) を参照してください。

## サポートされていない拡張

C++ コンパイラは『The Annotated C++ Reference Manual』(Ellis & Stroustrup 著、Addison-Wesley) に記載されている以下の一般的拡張をサポートしていません。

すべてのオブジェクトのメモリを一度に確保、解放する、`operator new[]` と `operator delete[]` メソッドのオーバーロードはサポートしていません。その代わり、`operator new[]` と `operator delete[]` が呼び出す `operator new()` と `operator delete()` をオーバーロードします (ARM、5.3.3、5.3.4)。

## C++ コンパイラの設定

ここでは、[C/C++ 言語設定パネル](#)のオプションの設定方法を説明します。この設定によって Metrowerks C++ の挙動が変わります。他のオプションについては、「[C/C++ 言語設定パネル](#)」(p15) を参照してください。

[C++ コンパイラを必ず利用](#)

[ARM に適合](#)

[C++ 例外処理有効](#)

[RTTI 有効](#)

[bool 型サポート](#)

[C++ 拡張の追加](#)

Direct to SOM に関する詳細は、『Targeting Mac OS』マニュアルを参照してください。

## C++ コンパイラを必ず利用

[C/C++ 言語設定パネル](#)の [ C++ コンパイラを必ず利用 ] オプションがオンの場合、コンパイラはプロジェクト内のすべてのCソースファイルをC++コードとしてコンパイルします。このオプションがオフの場合、CodeWarrior IDE はファイル名の拡張子を見て C または C++ コンパイラのどちらを使うか判断します。拡張子はファイルマッピング言語設定パネルで設定します。詳細は『IDE User Guide』を参照してください。

このオプションはプラグマ [cplusplus](#) に相当します。このプラグマが on か否かをチェックするには、`__option (cplusplus)` を使います。デフォルトでは、このオプションはオフです。

参照：[「cplusplus」\(p82\)](#)、[「オプションのチェック」\(p162\)](#)

## ARM に適合

[ ARM に適合 ] オプションがオンの場合、Metrowerks C++ が『The Annotated C++ Reference Manual』の C++ 言語仕様と衝突する ANSI/ISO C++ 機能を見つけるとエラーになります。このオプションは、『The Annotated C++ Reference Manual』の言語仕様に厳密に従わなければならないときだけ使ってください。

このオプションがオンの場合、以下のことができなくなります。

プロテクトされた基底クラスを使用する (ARM、11.2)

---

```
class X {};  
class Y : protected X {};
```

// CodeWarrior C++ では OK. ARM では Error

---

条件判断オペレータの文法で、2つ目と3つ目の代入表記を括弧でくくらない (K&R、A7.16)

---

```
i ? x=y : y=z    // CodeWarrior C++ では OK. ARM では Error  
i ? (x=y):(y=z) // ARM および CodeWarrior C++ で OK
```

---

if、while、switch 文の条件内で変数を宣言する (K&R、A9.4、A9.5)

---

```
while (int i=x+y) { /* ... */ }  
    // CodeWarrior C++ では OK、ARM では Error
```

---

このオプションをオンにすると、以下のことが可能になります。

for 文の条件式内で宣言された変数を、for 文以降で使用する (K&R、9.5)

---

```
for(int i=1; i<1000; i++) { /* ... */ }  
return i; // ARM では OK, CodeWarrior C++ では Error
```

---

このオプションはプラグマ [ARM\\_conform](#) に相当します。このプラグマが on か否かをチェックするには、`__option (ARM_conform)` を使います。デフォルトでは、このオプションはオフです。

参照：[「ARM\\_conform」\(p78\)](#)、[「オプションのチェック」\(p162\)](#)

## C++ 例外処理有効

ANSI/ISO 規格の `try`、`catch` 命令を使うならば、C/C++ 言語設定パネルの [C++ 例外処理有効] オプションをオンにしてください。それ以外の場合、小さなサイズで高速なコードを作るためにオフにしてください。

---

ヒント：PowerPlant で Mac OS プログラミングを行っている場合、このオプションをオンにしてください。PowerPlant は C++ 例外処理を利用します。

---

Metrowerks の ANSI/ISO C++ の例外処理メカニズムの実装についての詳細は、[「C++ 例外処理の使用」\(p57\)](#) を参照してください。

このオプションはプラグマ [exceptions](#) に相当します。このプラグマが on か否かをチェックするには、`__option (exceptions)` を使います。デフォルトでは、このオプションはオフです。

参照：[「exceptions」\(p94\)](#)、[「オプションのチェック」\(p162\)](#)

## RTTI 有効

Metrowerks C++ は、`dynamic_cast` や `typeid` オペレータを含む RTTI (Run-Time Type Information) をサポートしています。これらのオペレータを使うには、C/C++ 言語設定パネルの [RTTI 有効] オプションをオンにしてください。

オペレータの詳細は [「RTTI の使用」\(p57\)](#) を参照してください。

## bool 型サポート

標準 C++ の `bool` 型 (`true` または `false` を表わす) を使うときは、[bool 型サポート] オプションをオンにしてください。`bool`、`true`、`false` をキーワードとして認識すると問題がある場合は、このオプションをオフにしてください。

`bool` 型および `true` と `false` の値を有効にすることは、`typedef` および `#define` で定義することとは違います。C++ の `bool` 型は、ANSI/ISO C++ 規格で定義された別個の型です。これを別の型として処理できないソースコードは、正しくコンパイルされません。

CodeWarrior C++ は `bool` 型と `unsigned char` 型を同等に扱いますが、そうではないコンパイラがあります。このようなコンパイラでコンパイルできたソースコードを、CodeWarrior C++ コンパイラでコンパイルする場合 [ `bool` 型サポート ] オプションをオフにしなければエラーが発生します。

このオプションはプラグマ [bool](#) に相当します。このプラグマが `on` か否かをチェックするには、`__option (bool)` を使います。デフォルトでは、このオプションはオフです。

参照 : [「bool」\(p80\)](#)、[「オプションのチェック」\(p162\)](#)

## C++ 拡張の追加

C++ コンパイラにはその他にも拡張があります。プラグマ [cpp\\_extensions](#) を `on` にすると、コンパイラは ANSI C++ 規格に対し以下の拡張を行います。[C/C++ 言語設定パネル](#)にはこれに相当するオプションはありません。

プラグマが `on` の場合、コンパイラは ANSI/ISO C++ 規格に以下の拡張を行います。

### 匿名の構造体 (ARM、9)

---

```
#pragma cpp_extensions on
void foo()
{
    union {
        long      hilo;
        struct { short hi, lo; };          // 匿名の構造体
    };
    hi=0x1234;
    lo=0x5678;          // hilo==0x12345678
}
```

---

### メンバー関数への不適切なポインタ (ARM、8.1c)

---

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
    void (Foo::*ptmf1)() = &Foo::f;      // 常に OK
    void (Foo::*ptmf2)() = f;             // OK, cpp_exptensions が on の場合
}
```

---

このプラグマが `on` か否かをチェックするには、`__option (cpp_extensions)` を使います。デフォルトはオフです。

参照 : [「cpp\\_extensions」\(p83\)](#)、[「オプションのチェック」\(p162\)](#)



## C++ 例外処理の使用

[C/C++ 言語設定パネル](#)の [ C++ 例外処理有効 ] オプションがオンの場合、例外処理のための `try`、`catch` ステートメントを使えます。C++ 例外処理の詳細は「[C++ 例外処理有効](#)」(p55) を参照してください。

[ C++ 例外処理有効 ] オプションがオンの場合、CodeWarrior C/C++ コンパイラでコンパイルしたすべてのコードで例外を発することができます。以下の場合には例外を発することができません。

Macintosh Toolbox 関数呼び出し

[ C++ 例外処理有効 ] オプションがオフでコンパイルされたライブラリ

CodeWarrior 8 以前の CodeWarrior C/C++ コンパイラでコンパイルされたライブラリ

CodeWarrior Pascal またはそれ以外のコンパイラでコンパイルされたライブラリ

これらのうちのどれかで例外を発すると、`terminate()` が呼び出され終了します。

1 つまたは複数のクラスオブジェクトをアロケートしている最中に例外を発すると、部分的に構築されたオブジェクトは自動的にデストラクトされ、それらに割り当てられたメモリは解放されます。

## RTTI の使用

CodeWarrior C++ は、`dynamic_cast` や `typeid` オペレータを含む RTTI (Run-Time Type Information) をサポートしています。これらのオペレータを使うには、C/C++ 言語設定パネルの [ RTTI 有効 ] オプションをオンにしてください。

[dynamic\\_cast オペレータの使用](#)

[typeid オペレータの使用](#)

### dynamic\_cast オペレータの使用

`dynamic_cast` オペレータを使うと、ある型へのポインタを異なる型へのポインタに安全に変換することができます。通常のキャストと異なり、`dynamic_cast` は変換ができないときは 0 を戻します。通常のキャストでは、変換不可能なときにはプログラムをクラッシュさせかねない予測不可能な値が戻ります。

`dynamic_cast` オペレータの文法を示します。

---

```
dynamic_cast<Type*>(expr)
```

---

`Type` は、`void` または、少なくとも 1 つの仮想関数メンバーを持つクラスでなければいけません。`expr` が `(*expr)` で指し示すオブジェクトは、`Type` 型か、`Type` 型から派生されるものなら、これは、`expr` を `Type*` 型のポインタに変換しそれを戻します。そうでなければ、0 (ヌルポインタ) を戻します。

例えば、以下のクラスがあり、

---

```
class Person { virtual void func(void) { ; } };  
class Athlete : public Person { /* . . . */ };  
class Superman : public Athlete { /* . . . */ };
```

---

以下のポインタがあるとき、

---

```
Person *lois = new Person;  
Person *arnold = new Athlete;  
Person *clark = new Superman;  
Athlete *a;
```

---

`dynamic_cast` はそれぞれについて以下のように働きます。

---

```
a = dynamic_cast<Athlete*>(arnold);  
    // a は arnold, arnold は Athlete である  
a = dynamic_cast<Athlete*>(lois);  
    // a は 0, lois は Athlete ではない  
a = dynamic_cast<Athlete*>(clark);  
    // a は clark, clark は Superman かつ Athlete である
```

---

`dynamic_cast` オペレータは、リファレンス型と共に使うこともできます。しかし、リファレンスにはヌルポインタに相当するものがないので、変換ができないとき `dynamic_cast` は `bad_cast` 型の例外を起動します。

---

注意: `bad_cast` 型はヘッダファイル `exception` で定義されています。リファレンスに対して `dynamic_cast` を使うときは、必ず `#include <exception>` としてください。

---

以下は、リファレンスに対して `dynamic_cast` を使った例です。

---

```
#include <exception>  
// . . .  
Person &superref = *clark;  
  
try {  
    Person &ref = dynamic_cast<Person&>(superref);  
}  
catch(bad_cast) {  
    cout << "oops!" << endl;  
}
```

---

## typeid オペレータの使用

typeid オペレータを使うと、オブジェクトの型を知ることができます。typeid は sizeof オペレータのように、2 種類の引数を持つことができます。

クラス名

オブジェクトと評価される表記

---

注意: typeid オペレータを使うときは、必ず typeidinfo ヘッダファイルをインクルードしてください。

---

typeid オペレータは typeidinfo オブジェクトのリファレンスを戻します。これは、== や != で比較できます。以下のように、上からクラスを引用した場合、

---

```
class Person { /* . . . */ };
class Athlete : public Person { /* . . . */ };

Person *lois = new Person;
Athlete *arnold = new Athlete;
Athlete *louganis = new Athlete;
```

---

以下のすべての表記は正しいことになります。

---

```
#include <typeidinfo>
// . . .
if (typeid(Athlete) == typeid(*arnold))
    // arnold は Athlete である, 結果は true
if (typeid(*arnold) == typeid(*louganis))
    // arnold と louganis は Athletes である, 結果は true
if (typeid(*lois) == typeid(*arnold)) // ...
    // lois と arnold は同じタイプではない, 結果は false
```

---

typeidinfo クラスの name() メンバー関数で、型の名前を得ることができます。以下の例は、

---

```
#include <typeidinfo>
// . . .
cout << "Lois is a(n) "
      << typeid(*lois).name() << endl;
cout << "Arnold is a(n) "
      << typeid(*arnold).name() << endl;
```

---

以下のように印字されます。

---

```
Lois is a(n) Person
Arnold is a(n) Athlete
```

---

## テンプレートの使用

(ARM, 14) この節では、ファイル内でテンプレート宣言と定義をする最良の方法について説明します。また、ARM にはないが ANSI/ISO C++ にある文法を使って、どのようにテンプレートを明確にインスタンス化するのかについても説明します。

[テンプレートの宣言と定義](#)

[テンプレートのインスタンス化](#)

### テンプレートの宣言と定義

[例 4.8](#) のように、ヘッダファイル内でクラス関数と関数テンプレートを宣言します。

例 4.8      templ.h : テンプレート宣言ファイル

---

```
template <class T>
class Templ {
    T member;
public:
    Templ(T x) { member=x; }
    T Get();
};

template <class T>
T Max(T,T);
```

---

[例 4.9](#) のように、ソースファイル内でヘッダファイルをインクルードし、関数テンプレートとクラステンプレートのメンバー関数を定義します。

ソースファイルはテンプレート定義ファイルです。テンプレートを使うファイルすべてにこのファイルをインクルードします。テンプレート定義ファイルをプロジェクトに加える必要はありません。これは技術的にはソースファイルですが、ヘッダファイルの働きをします。

テンプレート定義ファイルはコードを生成しません。テンプレートの引数の値を指定しなければ、コンパイラはテンプレート用のコードを生成しません。これらの値を指定することを、テンプレートのインスタンス化と呼びます。[「テンプレートのインスタンス化」\(p62\)](#)を参照してください。

## 例 4.9      templ.cp : テンプレート定義ファイル

---

```
#include "templ.h"

template <class T>
T Templ<T>::Get()
{
    return member;
}

template <class T>
T Max(T x, T y)
{
    return ((x>y)?x:y);
}
```

---

**警告！** .h で終わるテンプレート宣言ファイルはインクルードしないでください。.h で終わるテンプレート宣言ファイルをソースファイルにインクルードすると、コンパイラはその関数もしくはクラスが定義されていないというエラーを發します。

---

## テンプレート宣言時に宣言を提供

CodeWarrior C++ は、テンプレートがインスタンス化されたときではなく、宣言されたときにテンプレート内の宣言を処理します。

現バージョンの C++ コンパイラは、テンプレート宣言時に利用できないテンプレート内の宣言を受け付けますが、これは将来改良されます。

## 例 4.10      テンプレート宣言の宣言

---

```
// Names in a class template declaration have to be defined

template<typename T> struct foo {
    bar *member; // illegal (but currently accepted)
};
struct bar { };
foo<int> fi;

// Workaround: Declare all names before using them:

struct bar;
template<typename T> struct foo {
    bar *member; // OK
};
struct bar { };
foo<int> fi;
```

```
// Names in template argument dependent base classes:

template<typename T> struct foo {
    typedef T *tptr;
};
template<typename T> struct bar : foo<T> {
    tptr member; // illegal (but currently accepted)
};

// Workaround: Use qualified name syntax:

template<typename T> struct foo {
    typedef T *tptr;
};
template<typename T> struct bar : foo<T> {
    typename foo<T>::tptr member; // OK
};

// The correct usage of typename in template argument
// dependent qualified names in some contexts:

template<class T> struct X {
    typedef X *xptra;
    xptra f();
};
template<class T> X<T>::xptra X<T>::f() // 'typename' missing
{
    return 0;
}

// Workaround: Use 'typename':

template<class T> typename X<T>::xptra X<T>::f() // OK
{
    return 0;
}
```

---

## テンプレートのインスタンス化

以下のことが行われていなければ、コンパイラはテンプレートのコードを生成することができません。

- テンプレートクラスを宣言する

- テンプレート定義を行う

- テンプレートのデータ型を指定する

最初の2つについては、[「テンプレートの宣言と定義」\(p60\)](#)を参照してください。

データ型とテンプレートの引数の指定は、テンプレートのインスタンス化と呼ばれます。Metrowerks C++ では、テンプレートをインスタンス化する2つの方法があります。最初に使われたときにコンパイラが自動的にインスタンス化する方法と、必要とされるすべてのインスタンス化を一箇所で明示的に作成する方法です。

#### 自動的インスタンス化

テンプレートを自動的にインスタンス化するには、そのテンプレートを使うすべてのソースファイルにテンプレート定義ファイルをインクルードし、異なる型や関数が必要な時にそのテンプレートを使うだけです。コンパイラは新しいものを見つける度に、テンプレートをインスタンス化したコードを自動的に作成します。[例 4.11](#) は、[例 4.8](#) と [例 4.9](#) にあるテンプレートを自動的にインスタンス化する方法を示しています。

#### 例 4.11 myprog.cp : テンプレートを使用するソースファイル

---

```
#include <iostreams.h>
#include "templ.cp" // includes templ.h as well

void main(void) {
    Templ<long> a = 1, b = 2;
    // コンパイラはここで Templ<long> をインスタンス生成
    cout << Max(a.Get(), b.Get());
    // コンパイラはここで Max<long>() をインスタンス生成
}
```

---

自動的にインスタンス化する場合、どのようなインスタンス化が必要であるのかをコンパイラ自身が判断しなければならないので、コンパイルに時間がかかります。また、そのテンプレートがインスタンス化されたオブジェクトコードがプログラム中に分散します。

#### 明示的インスタンス化

テンプレートを明示的にインスタンス化するには、テンプレート定義ファイルをソースファイル内に入れ、それぞれのインスタンス化のときに `template` というインスタンス化ステートメントを書きます。クラステンプレートインスタンス化の書式を以下に示します。

---

```
template class class-name<templ-specs>;
```

---

関数テンプレートインスタンス化の書式を以下に示します。

---

```
template return-type func-name<templ-specs>(arg-specs)
```

---

[例 4.12](#) では、[例 4.8](#) と [例 4.9](#) にあるテンプレートを明示的にインスタンス化する方法を示しています。

例 4.12 myinst.cp : テンプレートの明示的インスタンス化

---

```
#include "templ.cp"
```

```
template class Templ<long>; // クラスのインスタンス化  
template long Max<long>(long, long); // 関数のインスタンス化
```

---

関数を明示的にインスタンス化するときは、コンパイラが `arg-specs` から推測できる `templ-specs` の引数を書く必要はありません。例えば、[例 4.12](#) の `Max<long>()` はこのようになります。

---

```
template long Max<>(long, long);  
// コンパイラは引数から  
// Max<long>() のインスタンス化をしていることを判断する
```

---

明示的インスタンス化を使うと、コンパイルはより早く済みます。インスタンス化は 1 つのファイル内で成されるので、それらをすべて同じセグメントに入れたり、独立したライブラリにしたりすることができます。

---

注意： 明示的インスタンス化は ARM にはありませんが、ANSI/ISO C++ 規格の一部になっています。

---



## 第 5 章 C++ とエンベデッドシステム

この章では Metrowerks C/C++ を使ってエンベデッドシステム用ソフトウェアを作成する方法を説明します。C++ プログラムのサイズを小さくするヒントもあります。

### C++ とエンベデッドシステムの概要

ここでは以下の内容について説明します。

[EC++ 互換モード](#)

[ANSI/ISO C++ と EC++ の違い](#)

[CodeWarrior の EC++ 仕様](#)

[C++ コードのサイズ](#)

注意： この章はエンベデッドシステム用のプログラム設計の概念を説明するものです。現在 CodeWarrior C++ を使って EC++ (Embedded C++ 規格) 互換のエンベデッドシステムの開発が可能です。EC++ プロポーザルで述べられているライブラリのいくつかは CodeWarrior C++ に含まれていません。

### EC++ 互換モード

EC++ (Embedded C++) ソースコードをコンパイルするには、C/C++ 言語設定パネルの [ EC++ 互換モード ] オプションをオンにします。

コンパイル時に EC++ の互換性をテストするには、定義済みシンボル、[embedded\\_cplusplus](#) を使います。詳細は「[定義済みシンボル](#)」(p160) を参照してください。

### ANSI/ISO C++ と EC++ の違い

ANSI/ISO C++ (ANSI C++) の機能のいくつかは EC++ にはありません。EC++ では以下の機能はサポートされていません。

[テンプレート](#)

[ライブラリ](#)

[ファイル操作](#)

[ローカリゼーション](#)

## [例外処理](#)

## [その他の言語特性](#)

### テンプレート

ANSI C++ はテンプレートをサポートします。EC++ にはクラスや関数へのテンプレートサポートは含まれません。

### ライブラリ

クラス `<string>`、`<complex>`、`<ios>`、`<streambuf>`、`<istream>`、`<ostream>` は Embedded C++ 仕様書でサポートされています。テンプレート形式はサポートされません。その他の ANSI C++ ライブラリは (STL 型アルゴリズムライブラリを含めて) サポートされていません。

### ファイル操作

EC++ では、単純なコンソール入力や出力ファイルタイプ以外のファイル操作は指定されていません。

### ローカリゼーション

余分なメモリが必要になるため、EC++ にはローカリゼーションライブラリはありません。

### 例外処理

EC++ では例外処理はサポートされていません。

### その他の言語特性

以下の言語特性はサポートされていません。これ以外のマイナーな特性もサポートされていません。

記憶クラス指定子 ( `mutable` )

実行時型情報 ( RTTI )

名前空間

多重継承

仮想継承

## CodeWarrior の EC++ 仕様

EC++ との対応について説明します。EC++ 規格を満たすソフトウェア設計を異なる角度から説明します。

## [言語関係の問題](#)

## ライブラリ関係の問題

### 言語関係の問題

ソースコードを ANSI/ISO C++ および EC++ 規格の両方に対応させるためには、以下の条件を守ってください。

RTTI ( Run-Time Type Identification ) を使わない

サポートされていない機能拡張 ( 例外処理、名前空間など ) を使わない

多重継承や仮想継承を使わない

C++ 機能のいくつか ( RTTI や例外処理など ) は、コンパイラの設定パネルでオフにすることができます。C++ 言語設定パネルのオプションについては「[コンパイラ設定の概要 \( p15 \)](#)」を参照してください。

### ライブラリ関係の問題

C++ 用の MSL ( Metrowerks Standard Libraries ) のルーチン、データ構造体、クラスを参照しないでください。

Metrowerks は EC++ 対応アプリケーションに適したクラスライブラリを作成中です。次期バージョンでの発表を予定しています。

## C++ コードのサイズ

C++ においては、プログラムのサイズは非常に重要です。コードサイズを最適化するための方法がいくつかあります。

---

注意： オブジェクトのサイズを減らすことはコードパフォーマンスに大きな効果を与えます。

---

EC++ は仕様の一部としてある方法を使っています。その他の方法は C++ プログラミング一般に当てはまります。これらの方法は、EC++ 規格に準拠しているかどうかに関係なく、すべての C++ プログラムに当てはまります。

CodeWarrior では、コンパイラ、言語、ライブラリのグループにまとめてあります。

コンパイラに関連する方法

コンパイラに関連する方法では、オブジェクトコードのサイズを減らすというコンパイラの機能に依存します。

[サイズの最適化](#)：サイズを最適化するコンパイラのオプションを使います。

[インライン化](#)：インライン疑似命令の有効範囲をコントロールします。

### 言語に関連する方法

言語に関連する方法では、ANSI/ISO C++ の機能を制限するか、または使用しません。この方法によってソフトウェア設計とメンテナンスが簡単になりますが、コードサイズが増えることもあります。

[仮想関数](#)：仮想関数を使用しないことで、コードのサイズが減少します。

[RTTI](#)：プログラムが RTTI ( Runtime Type Identification ) を使っていないければ、コンパイラは余分なデータを生成しません。

[例外処理](#)：CodeWarrior C++ は実行速度を上げるためにオーバーヘッドのない例外処理を提供しますが、これは余分なオブジェクトコードを生成します。

[オペレータ new](#)：オペレータ new の内部で例外を投げ出さないように注意してください。

[多重継承](#)：多重継承が使われていないければ、コンパイラは余分なデータを生成しません。

### ライブラリに関連する方法

[ストリームベースクラス](#)：MSL に含まれるこれらのクラスは多くのコードを含んでいます。

[他のクラスライブラリを使う](#)：非標準のクラスライブラリでは、よりオーバーヘッドの少ない標準クラスライブラリの機能サブセットを提供しているかもしれません。

## サイズの最適化

CodeWarrior コンパイラにはサイズ、速度などの最適化レベルを設定するオプションがあります。

ターゲットごとの最適化はオプションでコントロールできます。これらのオプションはそれぞれのプロセッサ設定パネルにあります。

デバッグするときは、最適化をすべてオフにしてコンパイルしてください。ソースコードとオブジェクトコードの対応を壊す最適化もあるためです。デバッグが終わってからコードの最適化を行ってください。

[「コンパイラ設定の概要」\(p15\)](#) を参照してください。

## インライン化

CodeWarrior ではインラインをオフ、ノーマルインライン、自動インライン、最大インラインに設定することができます。

インライン化はコードサイズを減少、または増大します。この質問に対しての絶対的な答えはありません。小さい関数をインライン化すると、プログラムは小さくなります。たかさんの getter/setter メンバー関数を持つクラスライブラリをインライン化しても小さくなるのはほんの少しだけです。

MSL C++ は多数の関数をインライン定義していますので、コードサイズを最小にしたい場合には不適當です。MSL C++ を使っているときにコードサイズを最小にしたい場合、ライブラリをビルドするときにインラインをオフにします。MSL C++ を使っていないのならば、ノーマルインラインが適しています。

CodeWarrior では、ターゲット設定の言語オプションとしてインラインをコントロールできます。これらのオプションは C/C++ 言語設定パネルにあります。

コードをデバッグするときは、インラインをすべてオフにして、ソースコードとオブジェクトコードの対応を維持してください。デバッグが終わってからインライン化を行ってください。

[「関数のインライン」\(p38\)](#) を参照してください。

## 仮想関数

コードサイズを最適化するためには、必要以外に仮想関数は使わないでください。仮想関数は未使用であっても削除されません。

## RTTI

コードサイズが問題であれば、RTTI ( Runtime Type Identification ) は使わないでください。RTTI は各クラスにつき 2 倍のデータを生成します。RTTI をオフにするとデータセクションが小さくなります。

EC++ では RTTI を許可していません。C/C++ 言語設定パネルの [ RTTI 有効 ] オプションをオフにしてください。

[「RTTI 有効」\(p55\)](#) を参照してください。

## 例外処理

C++ の例外処理ルーチンを使う場合、[ C++ 例外処理有効 ] オプションをオンにしてください。オフの場合は例外処理を使わないでください。CodeWarrior にはランタイムオーバーヘッドのないエラー処理機構があります。しかし、例外を使うとコードサイズが増えます。例外テーブル ( データ ) は例外処理を使うとかなり大きくなります。

EC++ では例外処理を許可していません。C++ 言語設定パネルの [ C++ 例外処理有効 ] オプションオフにしてください。

---

注意： ANSI/ISO 規格ライブラリと new オペレータを使うには、例外処理が必要となります。[「オペレータ new」\(p69\)](#) を参照してください。

---

## オペレータ new

C++ のオペレータ new は、ランタイムライブラリのオペレータ new の実装によって例外を throw します。例外を throw するには、\_\_throws\_bad\_alloc を 1 にします。throw しな

いようにするには、ターゲットのプリフィックスファイルで `__throws_bad_alloc` を 0 にしてライブラリをビルドします。

詳細はリリースノートと各『Targeting』マニュアルを参照してください。

## 多重継承

コードとデータのオーバーヘッドには多重継承の実装が必要です。

EC++ では多重継承は許可されていません。

## 仮想継承

コードサイズの最適化するためには、仮想継承を使わないでください。通常、仮想基底クラスは複雑で、コンストラクタ関数やデストラクタ関数にコードを追加します。

EC++ では仮想継承は許可されていません。

## ストリームベースクラス

MSL C++ のストリームベースクラスは、いくつかの直接および間接オブジェクトのインスタンスを初期化します。コードサイズが重要な場合、ストリームベースクラスを使わないでください。ストリームベースクラスは標準入力 (`cin`)、標準出力 (`cout`)、標準エラー (`cerr`) を含みます。通常の入力および出力ルーチンに相当するキャラクタも多数あります。よほどの必要がない限り、標準の C の入力、出力関数を使ってください。

標準の C++ のストリームクラスの他にも整形フォーマットの文字列ストリームがあります。これは重いオーバーヘッドを発生させるため避けた方がよいでしょう。サイズが重要ならば、C の `sprintf` または `sscanf` を使ってください。

EC++ ではテンプレート化されたクラスや関数は許可されていません。MSL は ANSI/ISO 規格と対応しています。

## 他のクラスライブラリを使う

MSL C++ は ANSI/ISO の C++ 規格に基づいています。C++ 規格はテンプレートの使用を実装しており、これが初期オーバーヘッドを発生します。

このオーバーヘッドを避けるためには、独自のベクトル、文字列、あるいはよく利用する他のユーティリティクラスを工夫してください。NIH's ( National Institute of Health ) Class Library など、利用できるクラスライブラリは他にもあります。

他のクラスライブラリを使う場合、仮想継承、RTTI などがコードサイズを増やす原因であることに留意してください。

## 第 6 章 プラグマとシンボル

この章では、CodeWarrior C/C++ コンパイラで利用できるプラグマ (pragma) と定義済みシンボルについて説明します。

### プラグマとシンボルの概要

[C/C++ 言語設定パネル](#)で、コンパイラがどのようにプロジェクト全体をコンパイルするかを設定します。プラグマを使うことにより、コンパイラがどのようにコードをコンパイルするかを制御することもできます。

プラグマのほとんどは [C/C++ 言語設定パネル](#)、68K プロセッサ設定パネル、PPC プロセッサ設定パネルなどのオプションに相当します。

一般的に、大部分のコードに対しては設定パネルでオプションを設定し、特殊な場合にプラグマを使ってそのオプションを変更します。例えば、[C/C++ 言語設定パネル](#)で時間のかかる最適化をオフにして、プラグマを使って一番重要な部分のコードに対してのみオンにすることができます。

---

ヒント： MPW や Be OS 用の Metrowerks コマンドラインツールのオプションを使って #pragma 記述と同様の効果を得るには『Command-Line Tools』マニュアルを参照してください。

---

[プラグマの文法](#)：プラグマについて説明します。

[プラグマの有効範囲](#)：プラグマの有効範囲を説明します。

[プラグマ](#)：プラグマをリストにまとめてあります。

[定義済みシンボル](#)：ANSI と CodeWarrior のシンボルをまとめてあります。

[オプションのチェック](#)：プラグマとオプションの設定をテストする方法を説明します。

### プラグマ

ここではプラグマとその使い方について説明します。

[プラグマの文法](#)

[プラグマの有効範囲](#)

## プラグマの文法

プラグマのほとんどはこの文法で記述されます。

---

```
#pragma option-name on | off | reset
```

---

一般に、オプションの設定を変えるのに `on` か `off` を使い、オプションのオリジナルの設定に戻したいとき `reset` を使います。

---

```
#pragma profile off
// [ プロファイラ情報を生成 ] オプションがオンの場合、
// これに続く関数ではそれを off にする。

#include <smallfuncs.h>

#pragma profile reset
// [ プロファイラ情報を生成 ] オプションがオンの場合このオプションを
// on に戻す。そうでない場合は off のまま。
```

---

`#pragma profile reset` の代わりに `#pragma profile on` を使ったと仮定します。プロセッサ設定パネルで [ プロファイラ情報を生成 ] オプションをオフにすると、そのプラグマはオプションをオンにします。reset を使うことにより、設定パネルのオプション設定を不注意に変更しないで済みます。

## プラグマの有効範囲

一般的にプラグマは、1 つのファイルに対してのみ有効です。

[「プラグマの文法」\(p72\)](#) でも説明しましたが、プラグマの名前の後に `on` または `off` を付けて設定を変更します。希望の設定を終了するとそのプラグマ以降のコードすべてが、以下の状態になるまでその設定でコンパイルされます。

```
on、off、reset の設定が変更されたとき
ファイルの終点に達したとき
```

各ファイルの行頭で、コンパイラはプロジェクトまたはデフォルトの設定に戻ります。

プラグマの設定はプリコンパイルヘッダファイルにはストアされません。言い替えると、プリコンパイルヘッダの内部でコンパイラの設定を変更できますが、その変更はそのファイル内部でのみ有効です。プリコンパイルヘッダファイルの終点でアクティブな設定は失われ、再度設定しなくてはなりません。

プリコンパイルヘッダファイルで宣言されたアイテム（データや関数）の設定は、プリコンパイルヘッダファイルがインクルードされたときに保存され、再ストアされます。

以下の例では、変数 `xxx` を `far` 変数にしています。



```
// in file pch.h

#pragma far_data on
extern int xxx;
```

以下は、プリコンパイルヘッダとしてファイルをインクルードしていることを確認します。

```
// in file test.c
#pragma far_data off // far data is off

#include "pch.pch" // this file set far_data on

// far_data is still 'off' but xxx is still a far variable
```

プラグマ設定はヘッダファイル内部では有効ですが、ヘッダをインクルードしたソースファイルの設定は異なります。

a6frames

説明 A6 レジスタをベースにしたスタックフレームの生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma a6frames on | off | reset

注意 このプラグマは 68K Mac OS プログラミングでのみ有効です。

このプラグマが on の場合、コンパイラは A6 スタックフレームを作成し、デバッガでスタックコールのトレースや各ルーチンを探し出すことが可能になります。CodeWarrior デバッガや Jasik's The Debugger を含む多くのデバッガは、これらのフレームを必要とします。off の場合、コンパイラはこれらのフレームを作成しないので、生成したコードは小さくかつ速くなります。

このプラグマが on の場合、コンパイラが各関数に次のコードを作成します。

```
LINK #nn,A6
UNLK A6
```

このプラグマは 68K プロセッサ設定パネルの[ A6 スタックフレームを生成 ]オプションに相当します。on か否かをチェックするには、\_\_option (a6frames) を使います ([「オプションのチェック」\(p162\)](#))。

align

説明 データの配置方法を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma options align= alignment`

注意 このプラグマは構造体とクラスの配置方法を指定します。ここで、*alignment* は以下の値のいずれかになります。

<i>alignment</i>	コンパイラの処理
mac68k	フィールドが 1 バイト長ではない限り、すべてのフィールドを 2 バイト境界に置きます。68K Mac OS コンピュータで標準の配置です。
mac68k4byte	すべてのフィールドを 4 バイト境界に置きます。
power	すべてのフィールドを自然な境界に置きます。PowerPC Mac OS コンピュータで標準のアライメントです。例えば、char を 1 バイト境界、16 ビット int を 2 バイト境界に置きます。構造体を含む構造体や配列に対しても、このアライメントが再帰的に適用されます。従って、例えば 4 バイト浮動小数点メンバーをもつ構造体の配列は、4 バイト境界に置かれます。
native	すべてのフィールドを標準のアライメントに置きます。68K Mac OS コンピュータでは mac68k、PowerPC Mac OS コンピュータでは power を使うのと同じです。
packed	すべてのフィールドを 1 バイト境界に置きます。これは、どの設定パネルにもありません。このアライメントでは、コードがクラッシュしたり、多くのプラットフォームで遅く実行されたりします。注意して使用してください。
reset	直前に #pragma options align 記述があればその値に、そうでなければ 68K/PPC プロセッサ設定パネルで設定された値に戻します。

options と align の間にスペースがあることに注意してください。

このプラグマは、68K プロセッサ設定パネルの[ 構造体バイト配置 ]ポップアップメニューに相当します。

align\_array\_members

説明 struct および class データ内の配列の配置方法をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma align\_array\_members on | off | reset

注意 このプラグマは Mac OS プログラミングでのみ有効です。

このプラグマで、構造体やクラスの配列をどのように配置するのかわを選択できます。on の場合、コンパイラは 1 バイトよりも大きな配列フィールドを [ 構造体バイト配置 ] ポップアップメニューの設定に従って配置します。off の場合、コンパイラは配列フィールドを配置しません。

例 6.1     配列の配置を選択

```
#pragma align_array_members off
struct X1 {
    char c;           // offset==0
    char arr[4];      // offset==1 (char 配置)
};

#pragma align_array_members on
#pragma align mac68k
struct X2 {
    char c;           // offset==0
    char arr[4];      // offset==2 (2 バイト配置)
};

#pragma align_array_members on
#pragma align mac68k4byte
struct X3 {
    char c;           // offset==0
    char arr[4];      // offset==4 (4 バイト配置)
};
```

このプラグマが on か否かをチェックするには \_\_option (align\_array\_members) を使います (「[オプションのチェック](#)」(p162))。デフォルトは off です。

altivec\_codegen

説明 最適化に PowerPC AltiVec インストラクションを使う最適化を行います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma altivec_codegen on | off | reset`

注意 `on` の場合、可能であればコンパイラは PowerPC AltiVec インストラクションを最適化に使用します。

このプラグマが `on` か否かをチェックするには `__option (altivec_codegen)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトは `off` です。

`altivec_model`

説明 PowerPC AltiVec 言語拡張の使用を制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	<b>PowerPC</b>	NEC V800	Intel x86	MIPS
-----	----------------	----------	-----------	------

プロトタイプ `#pragma altivec_model on | off | reset`

注意 `on` の場合、コンパイラは一部の PowerPC プロセッサに装備されている AltiVec インストラクションを言語拡張を利用します。これが `on` の場合、`__ALTIVEC__` も定義されます。

このプラグマが `on` か否かをチェックするには `__option (altivec_model)` を使います ([「オプションのチェック」\(p162\)](#)、[「定義済みシンボル」\(p160\)](#))。

`altivec_vrsave`

説明 関数呼び出しの間に AltiVec レジスタを保存するか否かを制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	<b>PowerPC</b>	NEC V800	Intel x86	MIPS
-----	----------------	----------	-----------	------

プロトタイプ `#pragma altivec_vrsave on | off | reset | allon`

注意 `on` の場合、コンパイラは VRSave レジスタを設定するための追加命令を、関数の先頭と末尾に生成します。これによりどの AltiVec レジスタを保存するかを指定します。`allon` の場合、すべての AltiVec レジスタが保存されます。

このプラグマが `on` か否かをチェックするには `__option (altivec_vrsave)` を使います ([「オプションのチェック」\(p162\)](#))。

`always_inline`

説明 インラインされた関数の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma always\_inline on | off | reset

注意 このプラグマが on の場合、コンパイラは inline キーワードで宣言された関数をすべてインライン化しようとします。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには \_\_option (always\_inline) を使います ([「オプションのチェック」\(p162\)](#))。

ANSI\_strict

説明 基準にはない言語拡張の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma ANSI\_strict on | off | reset

注意 一般的な ANSI 拡張は以下のものです。プラグマ ANSI\_strict が on の場合、コンパイラは以下の拡張仕様のどれかを見つけるとエラーを出します。

C++ 形式のコメント

```
a = b;      // これは C++ 形式のコメント
```

関数定義内で名前のない引数

```
void f(int ) {} /* [ANSI に厳格に従う] オプションがオフなら OK */
void f(int i) {} /* 常に OK */
```

マクロ定義内の # に続く引数

```
#define add1(x) #x #1
/* [ANSI に厳格に従う] オプションがオフなら OK
   ただし、おそらく希望する通りではない。
   add1(abc) は "abc" #1 となる */

#define add2(x) #x "2"
/* 常に OK: add2(abc) は "abc2" となる */
```

#endif の後の識別子

```
#ifdef __MWERKS__          /* . . . */
#endif __MWERKS__          /* [ ANSI に厳格に従う ] オプションがオフなら OK
*/

#ifdef __MWERKS__          /* . . . */
#endif /*__MWERKS__*/      /* 常に OK */
```

このプラグマは [C/C++ 言語設定パネル](#) の [ ANSI に厳格に従う ] オプションに相当します。  
on か否かをチェックするには、\_\_option (ANSI\_strict) を使います (「[オプションの  
チェック](#)」(p162))。

### arg\_dep\_lookup

説明 C++ の引数に依存する名前の参照をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma arg\_dep\_lookup on | off | reset

注意 このプラグマが on の場合、コンパイラは引数に依存する名前を参照します。デフォルトは off です。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには \_\_option (arg\_dep\_lookup) を使います (「[オプションのチェック](#)」(p162))。

### ARM\_conform

説明 ARM 言語仕様以外の機能の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma ARM\_conform on | off | reset

注意 プラグマ ARM\_conform が on の場合、『The Annotated C++ Reference Manual』内の C++ 言語仕様とコンフリクトする ANSI C++ の機能が見つかったと、コンパイラはエラーを出します。このオプションは『The Annotated C++ Reference Manual』の言語仕様に厳密に従っているか否かを確認しなければいけないときだけ使ってください。

このプラグマが on の場合、次のことができなくなります。

プロテクトされた基底クラスを使用する。

---

```
class X {};  
class Y : protected X {}; //ARM_conform が off なら OK
```

---

条件オペレータの第 2、第 3 表記内に代入を使うとき、括弧なしの文法にする

---

```
i ? x=y : y=z // ARM_conform が off なら OK  
i ? (x=y):(y=z) // 常に OK
```

---

if、while、switch 文の条件内で変数を宣言する

---

```
while (int i=x+y) { . . . } //ARM_conform が off なら OK
```

---

このプラグマが on の場合、次のことが可能になります。

if 文の条件内で宣言された変数を、if 文以降で使う

---

```
for(int i=1; i<1000; i++) { /* . . . */ }  
return i; // ARM_conform が on なら OK
```

---

このプラグマは [C/C++ 言語設定パネル](#) の [ ARM に適合 ] オプションに相当します。on か否かをチェックするには、\_\_option (ARM\_conform) を使います([「オプションのチェック」\(p162\)](#))。

## auto\_inline

説明 インライン化する関数を選択します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma auto\_inline on | off | reset

注意 このプラグマが on の場合、コンパイラは自動的に関数をインライン化します。

[ インライン化しない ] オプション([「関数のインライン」\(p38\)](#))か、プラグマ dont\_inline ([「dont\\_inline」\(p92\)](#)) のどちらかがオンの場合、コンパイラはプラグマ auto\_inline を無視し、関数はインライン化されません。

このプラグマは [C/C++ 言語設定パネル](#) の [ 自動インライン展開 ] オプションに相当します。on か否かをチェックするには、\_\_option (auto\_inline) を使います([「オプションのチェック」\(p162\)](#))。

## bool

説明 bool、true、false をキーワードとして扱うか否かを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma bool on | off | reset`

注意 このプラグマが on の場合、true や false を表わす標準 C++ の bool 型を使えます。bool、true、false をキーワードとして認識すると問題になる場合、off にしてください。

このプラグマは [C/C++ 言語設定パネル](#) の [ bool 型サポート ] オプションに相当します(「[bool 型サポート](#)」(p55))。on か否かをチェックするには、`__option(bool)` を使います(「[オプションのチェック](#)」(p162))。デフォルトは off です。

## check\_header\_flags

説明 プリコンパイルヘッダのデータとプロジェクトのターゲット設定が一致しているかどうかを確認します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma check_header_flags on | off | reset`

注意 このプラグマはプリコンパイルヘッダにのみ有効です。

このプラグマが on の場合、プリコンパイル済みのヘッダの Double サイズ (8 バイトまたは 12 バイト)、int サイズ (2 バイトまたは 4 バイト)、浮動小数点演算の設定がビルドターゲットの設定と一致しているか否かをコンパイラが確認します。それらが対応していないと、コンパイラはエラーを出します。

プリコンパイル済みのヘッダファイルがプロジェクトと独立の設定値を持つとき、このプラグマを off にしてください。プリコンパイル済みのヘッダがそれらの設定に依存するとき、このプラグマを on にしてください。

このプラグマは、[C/C++ 言語設定パネル](#) のどのオプションとも対応していません。on か否かをチェックするには、`__option (check_header_flags)` を使います(「[オプションのチェック](#)」(p162))。デフォルトは off です。

## code\_seg

説明 コードを配置するセグメントを指定します。



**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma code_seg(name)`

**注意** このプラグマはコンパイル済みコードを格納するセグメントを指示します。*name* はコードセグメントの名前を指定する文字列です。以下に例を挙げます。

---

```
#pragma code_seg(".code")
```

---

これに続くコードはすべて `.code` という名前のコードセグメントに格納されます。

## code68020

**説明** Motorola 680x0（またはそれ以上の）プロセッサ向けのオブジェクトコード生成をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma code68020 on | off | reset`

**注意** このプラグマが `on` の場合、コンパイラは MC68020 向けに最適化したコードを生成します。このコードは Power Mac OS または MC68020 か MC68040 を持つ Mac OS で実行できます。MC68000 を持つ Mac OS ではクラッシュします。`off` の場合、コンパイラはどんな Mac OS でも実行可能なコードを生成します。

---

**警告！** 関数定義内でこのオプションの設定を変更しないでください。

---

MC68020 向けに最適化されたコードを実行する前に、`gestalt()` 関数を使ってそのチップが使えるかどうかを確認してください。`gestalt()` についての詳細は『Inside Mac OS: Operating System Utilities』の「Gestalt Manager」の章を参照してください。

Mac OS 用コンパイラでは、このオプションはデフォルトではオフです。

このプラグマは 68K プロセッサ設定パネルの [ 68020 コード生成 ] オプションに相当します。`on` か否かをチェックするには、`__option (code68020)` を使います ([「オプションのチェック」\(p162\)](#))。

## code68881

説明 Motorola 68881 (またはそれ以上の) 演算プロセッサ向けのオブジェクトコード生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma code68881 on | off | reset`

注意 このプラグマは 68K Mac OS プログラミングにのみ有効です。

このプラグマが `on` の場合、コンパイラは MC68881 浮動小数点ユニット (FPU) 向けに最適化したコードを生成します。このコードは、MC68881 FPU、MC68882 FPU、または MC68040 プロセッサを持つ Mac OS で実行可能です (MC68040 は MC68881 FPU を内蔵しています)。このコードは、Power Mac OS、MC68LC040 を持つ Mac OS、他のプロセッサで FPU を持たない Mac OS では実行できません。`off` の場合、コンパイラはどの Mac OS でも実行可能なコードを生成します。

**警告!** プラグマ `code68881` を使ってこのオプションを `on` にする場合、ファイルの先頭 (インクルードファイルや変数、関数宣言より前) に置いてください。

MC68881 向けに最適化されたコードを実行する前に、`gestalt()` 関数を使って FPU が使えるかどうかを確認してください。`gestalt()` についての詳細は、『Inside Mac OS: Operating System Utilities』の「Gestalt Manager」の章を参照してください。

このプラグマは 68K プロセッサ設定パネルの [ 浮動小数点モデル ] ポップアップメニューの [ 68881 ] オプションに相当します。`on` か否かをチェックするには、`__option (code68881)` を使います ([「オプションのチェック」\(p162\)](#))。

## cplusplus

説明 これ以降のソースコードを C または C++ ソースコードとしてコンパイルします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma cplusplus on | off | reset`

注意 このプラグマが `on` の場合、コンパイラはこれ以降のコードを C++ コードとみなしてコンパイルします。`off` の場合、コンパイラはファイル名の拡張子でどのようにコンパイルするかを判断します。ファイル名が `.cp`、`.cpp`、`.c++` で終わるとき、コンパイラはそれら

を自動的に C++ コードとしてコンパイルします。ファイル名が `.c` で終わるとき、コンパイラはそれらを自動的に C コードとしてコンパイルします。このプラグマは、ファイル内に C と C++ のコードが混在しているときだけ必要になります。

このプラグマは C/C++ 言語設定パネルの [ C++ コンパイラを必ず利用 ] オプションに相当します。on か否かをチェックするには、`__option (cplusplus)` を使います ( [「オプションのチェック」\(p162\)](#) )。

## cpp\_extensions

説明 ANSI/ISO C++ 規格の言語拡張を行います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma cpp_extensions on | off | reset`

注意 このプラグマが on の場合、ANSI C++ 規格が次のように拡張されます。

### 匿名の構造体

```
#pragma cpp_extensions on
void foo()
{
    union {
        long hilo;
        struct { short hi, lo; }; // 匿名構造体
    };
    hi=0x1234;
    lo=0x5678; // hilo==0x12345678
}
```

### メンバー関数を指す無条件ポインタ

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
    void (Foo::*ptmf1)() = &Foo::f; // 常に OK

    void (Foo::*ptmf2)() = f; // cpp_extensions が on ならば OK
}
```

このプラグマは設定パネルのどのオプションとも対応していません。on か否かをチェックするには、`__option (cpp_extensions)` を使います(「[オプションのチェック \(p162\)](#)」)。デフォルトは off です。

d0\_pointers

説明 どのレジスタに関数結果のポインタを入れるかを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma d0_pointers`

注意 このプラグマは 68K プログラミングにのみ有効です。

このプラグマで、2 つの呼び出し規約 (MPW と Mac OS Toolbox ルーチンの呼び出し規約と Metrowerks C/C++ の呼び出し規約) を選択できます。MPW と Mac OS Toolbox ルーチンの呼び出し規約では、関数はポインタを D0 レジスタに入れて戻します。Metrowerks C/C++ の呼び出し規約では、関数はポインタを A0 レジスタに入れて戻します。

Mac OS Toolbox 関数を宣言するときや MPW でコンパイルされたライブラリを使うときは、プラグマ `d0_pointers` を on にしてください。それらの関数を宣言した後、Metrowerks C/C++ 関数の宣言を開始するために、このプラグマを off にしてください。

[例 6.2](#) で、`Sound.h` 内の Toolbox 関数はポインタを D0 で戻し、`Myheader.h` 内のユーザー定義関数はポインタを A0 で戻します。

例 6.2 `#pragma pointers_in_A0` と `#pragma pointers_in_D0` の使用

```
#pragma d0_pointers on      //Toolbox コール用にセット
#include <Sound.h>
#pragma d0_pointers reset // ユーザー定義関数用にセット
#include "Myheader.h"
```

プラグマ `pointers_in_A0` と `pointers_in_D0` は、`d0_pointers` とほとんど同じ意味で、下位互換性確保のために使えるようになっています。プラグマ `pointers_in_A0` は `#pragma d0_pointers off` に、プラグマ `pointers_in_D0` は `#pragma d0_pointers on` に相当します。新規のコードには、`reset` 引数があるのでプラグマ `d0_pointers` の使用を推奨します。詳しくは「[pointers\\_in\\_A0, pointers\\_in\\_D0 \(p128\)](#)」を参照してください。

このプラグマは設定パネルのどのオプションとも対応していません。on か否かをチェックするには、`__option (d0_pointers)` を使います (「[オプションのチェック \(p162\)](#)」)。

## data\_seg

説明 このプラグマは無視されますが、Microsoft との互換性確保のために含まれています。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma data_seg(name)`

注意 無視されます。Microsoft との互換性確保のために含まれています。初期化されたデータを格納するセグメントを指定します。*name* はデータセグメントを指定する文字列です。以下に例を挙げます。

---

```
data_seg(".data")
```

---

この後に続くデータはすべて `.data` という名前のセグメントに格納されます。

## def\_inherited

説明 `inherited` キーワードの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma def_inherited on | off | reset`

注意 これにより C++ プログラミングで `inherited` キーワードが使用できます。デフォルトは `off` です。

---

注意: `inherited` キーワードは ANSI/ISO C++ 規格ではサポートされていません。CodeWarrior C++ では 1 回の継承だけが実装されています。

---

このプラグマが `on` か否かをチェックするには `__option (def_inherited)` を使います ([「オプションのチェック」\(p162\)](#))。

## defer\_codegen

説明 コンパイルされていない関数のインライン化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma defer_codegen on | off | reset`

注意 このプラグマにより、その定義よりも先に呼び出されるインラインまたは自動インライン関数のインラインが可能になります。

```
#pragma defer_codegen on
#pragma auto_inline on

extern void f();
extern void g();

main()
{
    f(); // インラインされる
    g(); // インラインされる
}

inline void f() {}
void g() {}
```

注意： このプラグマが `on` の場合、コンパイラはコンパイル時により多くのメモリを必要とします。

このプラグマは C/C++ 言語設定パネルの [ 定義前のインライン展開を許可 ] オプションに相当します (「[関数のインライン](#)」(p38))。このプラグマが `on` か否かをチェックするには `__option (defer_codegen)` を使います (「[オプションのチェック](#)」(p162))。

## define\_section

説明 オブジェクトコードをセクションへ並べ換えます。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma define_section sname istr [ustr] [addrmode] [accmode]`

注意 このプラグマによりコンパイルしたオブジェクトコードを定義済みのセクションや自分で定義したセクションへ配置することができます。

パラメータ：

*sname* : ユーザー定義セクションをソースから参照するための識別子。

例 :

```
#pragma section sname begin
```

または

```
__declspec(sname)
```

*istr* : *sname* に割り当てられた initialized データのセクション名文字列。  
.data など (*ustr* がない場合、未初期化データに適用される)

*ustr* : *sname* に割り当てられた uninitialized データの elf セクション名。

*addrmode* : セクションのアドレス方法を示す。以下のいずれかになる。

- standard : 32 ビット絶対アドレス
- near\_absolute : 16 ビット絶対アドレス
- far\_absolute : 32 ビット絶対アドレス
- near\_code : TP からの 16 ビットオフセット
- far\_code : TP からの 32 ビットオフセット
- near\_data : GP からの 16 ビットオフセット
- far\_data : GP からの 32 ビットオフセット

*accmode* : セクションの属性を示す。以下のいずれかになる。

- R : 読み取り可能 (readable)
- RW : 読み取り、書き込み可能 (readable and writable)
- RX : 読み取り、実行可能 (readable and executable)
- RWX : 読み取り、書き込み、実行可能 (readable, writable, executable)

*ustring* のデフォルト値は *istring* と同じです。*addrmode* のデフォルト値は standard です。*accmode* のデフォルト値は RWX です。

コンパイラは以下のコモン MIPS セクションを絶対アドレスモードで事前に定義します( [表 6.1](#) )

表 6.1 事前に定義された MIPS セクション

```
#pragma define_section text ".text" far_absolute RX
#pragma define_section data ".data" ".bss" far_absolute RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_absolute R
```

以下は C++ の実装のために予約されています ( [表 6.2](#) )

表 6.2 C++ 用の事前に定義された MIPS セクション

```
#pragma define_section exception ".exception" far_absolute R
#pragma define_section exceptlist ".exceptix" far_absolute R
#pragma define_section vtables ".vtables" far_absolute R
```

コンパイラは以下のコモン MIPS セクションを PID アドレスモードで事前に定義します( [表 6.3](#) )

表 6.3 PID アドレスモードで事前に定義された MIPS セクション

```
#pragma define_section text ".text" far_absolute RX
#pragma define_section data ".data" ".bss" far_data RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_absolute R
#pragma define_section exception ".exception" far_data R
#pragma define_section exceptlist ".exceptix" far_data R
#pragma define_section vtables ".vtables" far_data R
```

コンパイラは以下のコモン MIPS セクションを PIC アドレスモードで事前に定義します( [表 6.4](#) )

表 6.4 PIC アドレスモードで事前に定義された MIPS セクション

```
#pragma define_section text ".text" far_code RX
#pragma define_section data ".data" ".bss" far_absolute RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
#pragma define_section const ".rodata" far_code R
#pragma define_section exception ".exception" far_absolute R
#pragma define_section exceptlist ".exceptix" far_absolute R
#pragma define_section vtables ".vtables" far_absolute R
```

コンパイラは以下のコモン V810/V830 セクションを事前に定義します ( [表 6.5](#) )

表 6.5 NEC V810 および V830 プロセッサ用に事前に定義されたセクション

```
#pragma define_section text ".text" far_code RX
#pragma define_section data ".data" ".bss" far_data RW
#pragma define_section sdata ".sdata" ".sbss" near_data RW
```



```
#pragma define_section itext ".itext" far_absolute RX
#pragma define_section const ".const" far_absolute R
#pragma define_section sconst ".sconst" near_absolute R
#pragma define_section sedata ".sedata" ".sebss" near_absolute RW
#pragma define_section sidata ".sidata" near_absolute RW
#pragma define_section cdata1 ".cdata1" far_absolute RW
#pragma define_section cdata2 ".cdata2" far_absolute RW
#pragma define_section cdata3 ".cdata3" far_absolute RW
#pragma define_section udata1 ".udata1" far_absolute RW
#pragma define_section udata2 ".udata2" far_absolute RW
#pragma define_section udata3 ".udata3" far_absolute RW
```

以下は C++ の実装のために予約されています ( [表 6.6](#) )。

表 6.6 NEC V810 および V830 プロセッサ用に事前に定義されたセクション (C++)

```
#pragma define_section exception ".exception" far_data R
#pragma define_section exceptlist ".exceptlist" far_data R
#pragma define_section vtables ".vtables" far_data R
#pragma define_section string ".string" far_data RW
#pragma define_section cstring ".cstring" far_data R
```

`#pragma define_section` を使ってこれらの既存セクションの属性を再定義することができます。

1. すべてのデータに 16 ビット絶対アドレスを強制するには以下のようにします。

```
#pragma define_section data ".data" near_absolute
```

2. 例外テーブルに 32 ビット TP 相対アドレスを強制するには以下のようにします。

```
#pragma define_section exceptlist ".exceptlist" far_code
#pragma define_section exception ".exception" far_code
```

アドレスの強制をする場合、プリフィックスファイルか、プログラムの全ソースファイルにインクルードされるヘッダファイルにこれらの `#pragmas` を置くとよいでしょう。

注意： ユーザー定義セクションは ELF リンカの Section Mappings 設定パネルで適切なセグメントにマップしなくてはなりません。

NEV V800 :near\_absolute 定義されたセクションは 00000000:00007FFF から FFFF8000:FFFFFFFF の範囲内にあるセグメントに割り当てなくてはなりません。  
near\_codeまたはfar\_code定義されたセクションは .textと同じセグメントに割り当てなくてはなりません。near\_data または far\_data 定義されたセクションは .data と同じセグメントに割り当てなくてはなりません。

direct\_destruction

このプラグマはもう使用できません。

direct\_to\_som

説明 SOM オブジェクトコードの生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma direct\_to\_som on | off | reset

注意 このプラグマは C++ プログラミングでのみ有効です。

このプラグマにより、CodeWarrior IDE 内で直接 SOM コードを作成できます。SOM は OpenDoc に欠くことのできない一部です。詳しくは『Targeting Mac OS』マニュアルを参照してください。

このプラグマが on の場合、CodeWarrior C/C++ は [C/C++ 言語設定パネル](#)の [ Enum は常に Int 型 ] オプションを自動的にオンにします (「[enum は常に Int 型](#)」(p31))。

このプラグマは、C/C++ 言語設定パネルの [ Direct to SOM ] ポップアップメニューに相当します。このメニューの [ 利用する ] は、このプラグマを on にして SOMCheckEnviornment を off にすることと等価です。[ 環境チェックと共に利用する ] は、このプラグマと SOMCheckEnviornment の両方を on にすることと等価です。[ 利用しない ] は、このプラグマと SOMCheckEnviornment の両方を off にすることと等価です。  
SOMCheckEnviornment の詳細は「[SOMCheckEnvironment](#)」(p142)を参照してください。

このプラグマが on か否かをチェックするには、\_\_option (direct\_to\_SOM) を使います (「[オプションのチェック](#)」(p162))。デフォルトではこのプラグマは off です。

disable\_registers

説明 ANSI/ISO 関数の setjmp() の互換性をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma disable_registers on | off | reset`

注意 このプラグマが `on` の場合、コンパイラは `setjmp()` を呼ぶすべての関数の、ある種の最適化を無効にします。グローバル最適化も無効になり、局所変数や引数はレジスタにストアされません。これにより、すべての局所変数は最新の値をもつことになります。

注意： このオプションは、PowerPlant の TRY/CATCH マクロを使う関数のレジスタ最適化を無効にしますが、ANSI 標準の `try/catch` を使う関数では無効になりません。TRY/CATCH マクロは `setjmp()` を使っていますが、`try/catch` はより低いレベルで実装されており `setjmp()` を使っていません。

これは THINK C や Symantec C++ にある機能を模倣するプラグマです。この機能に依存しているコードを移植するときだけ使ってください。このプラグマはコードサイズの大幅な増加と実行速度の極端な低下を招くことになります。新規のコードでは、`setjmp()` コールの前後で値が変わらないようにしたいときは、その変数を `volatile` 宣言してください。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (disable_registers)` を使います([「オプションのチェック \(p162\)」](#))。デフォルトは `off` です。

## dollar\_identifiers

説明 ドル記号 (`$`) を識別子として使用可能にします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma dollar_identifiers on | off | reset`

注意 このプラグマが `on` の場合、コンパイラはドル記号 (`$`) を識別子として受け付けます。`off` の場合、コンパイラはアンダースコア、アルファベット、数字以外の識別子を見つけるとエラーを発します。

このプラグマのデフォルトは `off` です。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (dollar_identifiers)` を使います([「オプションのチェック \(p162\)」](#))。

## dont\_inline

説明 インライン関数の生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma dont_inline on | off | reset`

注意 プラグマ `auto_inline` が `on` の場合、関数が `inline` キーワードで宣言されていたり、クラス宣言内でメンバー関数が定義されていても、コンパイラは関数コールをインライン化しません。また、プラグマ `auto_inline` ([「auto\\_inline」\(p79\)](#)) の設定に関係なく、関数コールを自動的にインライン化しません。 `off` の場合、コンパイラはすべてのインライン関数コールを展開します。

このプラグマは C/C++ 言語設定パネルの[ インラインの深さ ]ポップアップメニューの[ インライン化しない ] に相当します。 `on` か否かをチェックするには、 `__option` (`dont_inline`) を使います ([「オプションのチェック」\(p162\)](#))。

## dont\_reuse\_strings

説明 各文字列の文字を別々のプールにストアします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma dont_reuse_strings on | off | reset`

注意 プラグマ `dont_reuse_strings` が `on` の場合、コンパイラはそれぞれの文字列の文字を別々にストアします。 `off` の場合、コンパイラは同一の文字列の文字を 1 つだけストアします。 変更する予定のない同一の文字列がたくさんあるとき、このプラグマによってメモリの節約ができます。

次のコードセグメントを見ると、

```
char *str1="Hello";
char *str2="Hello"
*str2 = 'Y';
```

このプラグマが `on` の場合、`str1` は "Hello" で `str2` は "Yello" です。 `off` の場合、`str1`、`str2` は両者とも "Yello" です。

このプラグマは C/C++ 言語設定パネルの [ 文字列定数を再利用しない ] オプションに相当します。プラグマが on か否かをチェックするには `__option (dont_reuse_strings)` を使います (「[オプションのチェック](#)」(p162))。

ecplusplus

説明 Embedded C++ 機能の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma ecplusplus on | off | reset`

注意 このプラグマが on の場合、C++ コンパイラは、ANSI C++ の EC++ 以外の機能 (テンプレート、多重継承など) を無効にします。CodeWarrrior C/C++ の embedded C++ のサポートについては「[C++ とエンベデッドシステム](#)」(p65) を参照してください。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには `__option (ecplusplus)` を使います (「[オプションのチェック](#)」(p162))。このプラグマのデフォルトは off です。

EIPC\_EIPSW

説明 割り込み関数のプロセッサ情報の保存をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma EIPC_EIPSW on|off|reset`

注意 割り込み関数のコンパイル中にこのプラグマが on になっている場合、コンパイラは EIPC および EIPSW を保存、リストアします。`__EIEP()` を呼び出してさらに割り込みを行っても安全です。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには `__option (dont_reuse_strings)` を使います (「[オプションのチェック](#)」(p162))。

enumsalwaysint

説明 enum 型のサイズを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma enumsalwaysint on | off | reset

**注意** プラグマ `enumsalwaysint` が `on` の場合、C/C++ コンパイラは `enum` 型を `int` 型と同じサイズにします。`enum` 定数が `int` よりも大きいとき、コンパイラはエラーを出します。`off` の場合、コンパイラは `enum` 型を任意の大きさの整数型にします。最も大きな `enum` 定数の大きさに最も近いサイズの `int` 型が選ばれます。その型は、最小で `char`、最大で `long int` です。

---

```
enum SmallNumber { One = 1, Two = 2 };
/* enumsalwaysint が on の場合、このタイプは char と同じサイズになる
   pragma が off の場合、このタイプは int と同じサイズになる */

enum BigNumber
{ ThreeThousandMillion = 3000000000 };
/* enumsalwaysint が on の場合、このタイプは long int と同じサイズになる
   pragma が off の場合、コンパイラがエラーを出す */
```

---

コンパイラが `enum` 型を処理する方法については「[enum は常に Int 型](#)」(p31)を参照してください。

このプラグマは C/C++ 言語設定パネルの [ Enum は常に Int 型 ] オプションに相当します。`on` か否かをチェックするには、`__option (enumsalwaysint)` を使います(「[オプションのチェック](#)」(p162))。

## exceptions

**説明** C++ 例外処理機能をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma exceptions on | off | reset

**注意** このプラグマが `on` の場合、例外処理に `try`、`catch` ステートメントが使えます。例外処理を使わない場合は、プログラムを小さくするために `off` にしておきます。

CodeWarrior 8 およびそれ以降の C/C++ 言語設定パネルで [ C++ 例外処理有効 ] オプションをオンにしてコンパイルされたすべてのコードを通じて、例外を発することができます。以下の場合には例外を発することができません。

Macintosh Toolbox 関数コール

[ C++ 例外処理有効 ] オプションがオフでコンパイルされたライブラリ

CodeWarrior 8 以前の CodeWarrior C/C++ コンパイラでコンパイルされたライブラリ

Metrowerks Pascal または他のコンパイラでコンパイルされたライブラリ

これらのケースのうちのどれかで例外を発すると、`terminate()` が呼ばれて終了します。

このプラグマは C/C++ 言語設定パネルの [ C++ 例外処理有効 ] オプションに相当します。  
on か否かをチェックするには、`__option (exceptions)` を使います ([「オプションのチェック」\(p162\)](#))。

## export

説明 モジュールからエクスポートするアイテムをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma export on | off | reset | list names`

注意 このプラグマは Mac OS プログラミングでのみ有効です。

プラグマ `export` を使うと、`.exp` ファイルを使わずにシンボルをエクスポートすることができます。このプラグマでシンボルをエクスポートするには、PPC PEF または CFM68K 設定パネルの [ シンボルをエクスポート ] ポップアップメニューから [ #pragma を使用 ] を選んでください。そして、このファイル内で宣言、定義された変数や関数をエクスポートするためにこのプラグマを `on` にします。[ シンボルをエクスポート ] ポップアップメニューから [ #pragma を使用 ] 以外のオプションを選ぶと、コンパイラはこのプラグマを無視します。

ある範囲内で宣言、定義されているすべての関数や変数をエクスポートしたいときは、その範囲の先頭で `#pragma export on` を使い、最後で `#pragma export off` を使います。リスト内のすべての関数や変数をエクスポートしたいときは、`#pragma export list` を使います。変数または関数を 1 つだけエクスポートする場合、宣言の先頭に `__declspec (export)` を置きます。

次のコードでは、変数 `w` と関数 `a1()`、`b1()` をエクスポートするとき、`#pragma export on` と `off` を使っています。

```
#pragma export on
int a1(int x, double y);
double b1(int z);
int w;
#pragma export off
```

次のコードでは、シンボルをエクスポートするのに `#pragma export list` を使っています。

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma export list a1, b1, w
```

次のコードでは、シンボルをエクスポートするのに `__declspec(dllexport)` を使っています。

```
__declspec(dllexport) int a1(int x, double y);
__declspec(dllexport) double b1(int z);
__declspec(dllexport) int w;
```

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option(dllexport)` を使います ([「オプションのチェック」\(p162\)](#))。

## extended\_errorcheck

説明 潜在的な論理的エラーに対して警告を発します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma extended_errorcheck on | off | reset`

注意 プラグマ `extended_errorcheck` が on の場合、C コンパイラは以下のいずれかを見つけると警告（エラーではない）を出します。

`return` 文を含まない `void` ではない関数。以下は警告になります。

```
main() /* return int を想定 */
{
    printf ("hello world\n");
} /* WARNING: return ステートメントがない */
```

以下は OK です。

```
void main()
{
    printf ("hello world\n");
}
```



enum 型に整数値や浮動小数点値を代入する。

```
enum Day { Sunday, Monday, Tuesday,
           Wednesday, Thursday,
           Friday, Saturday } d;

d = 5;           /* WARNING */
d = Monday;      /* OK */
d = (Day)3;      /* OK */
```

注意： これらは両者とも C++ ではエラーになります。

C/C++ コンパイラは以下のいずれかを見つけると警告を出します。

void 宣言されていない関数に空の return 文 (return;) がある場合。以下は警告になります。

```
int MyInit(void)
{
    int err = GetMyResources();
    if (err!=0) return; // WARNING: 空の return ステートメント

    // . . .
}
```

以下は OK です。

```
int MyInit(void)
{
    int err = GetMyResources();
    if (err!=0) return -1; // OK

    // . . .
}
```

このプラグマは C/C++ 警告設定パネルの[ 拡張エラーチェック ]オプションに相当します。on か否かをチェックするには、\_\_option (extended\_errorcheck) を使います ([「オプションのチェック」\(p162\)](#))。

far\_code, near\_code, smart\_code

説明 実行可能コードのアドレッシングの種類を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     

```
#pragma far_code,
#pragma near_code,
#pragma smart_code
```

注意 このプラグマは 68K Mac OS プログラミングでのみ有効です。

これらのプラグマは、関数を参照するためにコンパイラが使うアドレッシングを指定します。

`#pragma far_code` : 16 ビットアドレッシングが使用可能でも常に 32 ビットアドレッシングが生成されます。

`#pragma near_code` : データまたは命令が範囲外でも常に 16 ビットアドレッシングが生成されます。

`#pragma smart_code` : 使用可能ならば 16 ビットアドレッシングが、必要に応じて 32 ビットアドレッシングが生成されます。

これらのコードモデルについての詳細は『IDE User Guide』を参照してください。

これらのプラグマは 68K プロセッサ設定パネルの [ コードモデル ] ポップアップメニューに相当します。デフォルトは `#pragma smart_code` です。

## far\_data

説明 グローバルデータを参照するために 32 ビットアドレッシングを使います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     

```
#pragma far_data on | off | reset
```

注意 このプラグマが `on` の場合、グローバルデータは 16 ビットアドレッシングではなく 32 ビットアドレッシングで参照されるので、グローバルデータをいくつでも持つことが可能です。プログラムは、多少大きく、遅くなります。`off` の場合、グローバルデータは `near data` としてストアされ、64K の制限が適用されます。

このプラグマは 68K プロセッサ設定パネルの [ Far データ ] オプションに相当します。`on` か否かをチェックするには、`__option (far_data)` を使います ([「オプションのチェック」\(p162\)](#))。

## far\_strings

説明 文字列を参照するために 32 ビットアドレッシングを使います。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma far_strings on | off | reset`

**注意** このプラグマが `on` の場合、文字列は 16 ビットアドレッシングではなく 32 ビットアドレッシングで参照されるので、文字列をいくつでも持つことが可能です。プログラムは、多少大きく、遅くなります。`off` の場合、文字列は `near data` としてストアされ、64K の制限が適用されます。

このプラグマは 68K プロセッサ設定パネルの [ Far 文字列定数 ] オプションに相当します。`on` か否かをチェックするには、`__option (far_strings)` を使います ([「オプションのチェック」\(p162\)](#))。

### `far_vtables`

**説明** C++ 仮想関数テーブルに 32 ビットアドレッシングを使います。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma far_vtables on | off | reset`

**注意** このプラグマは 68K Mac OS プログラミングでのみ有効です。

仮想関数メンバーを持つクラスは、仮想関数ディスパッチテーブルをデータセグメントに持たなければいけません。このプラグマが `on` の場合、そのテーブルは 16 ビットアドレッシングではなく 32 ビットアドレッシングで参照されるので、どんな大きさでも許されます。プログラムは、多少大きく、遅くなります。`off` の場合、そのテーブルは `near data` としてストアされ、64K の制限が適用されます。

このプラグマは 68K プロセッサ設定パネルの [ Far メソッドテーブル ] オプションに相当します。`on` か否かをチェックするには、`__option (far_vtables)` を使います ([「オプションのチェック」\(p162\)](#))。

### `faster_pch_gen`

**説明** プリコンパイルヘッダの生成パフォーマンスをコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma faster_pch_gen on | off | reset`

注意 このプラグマが `on` の場合、プリコンパイルヘッダを高速に記述することができます(ヘッダの構造によります)。 `on` にして作成したプリコンパイルヘッダファイルのサイズは若干大きくなります。デフォルトは `off` です。

このプラグマに相当する設定パネルのオプションはありません。 `on` か否かをチェックするには `__option (faster_pch_gen)` を使います ([「オプションのチェック」\(p162\)](#))。

## float\_constants

説明 浮動小数点定数の扱い方をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma float_constants on | off | reset`

注意 このプラグマが `on` の場合、コンパイラは指定されていない浮動小数点定数の値を `float` 型として処理します。 `double` ではありません。

これは AMD K6 プロセッサ用にコンパイルしたソースコードを扱うときに便利です。

このプラグマに相当する設定パネルのオプションはありません。 `on` か否かをチェックするには `__option (float_constants)` を使います ([「オプションのチェック」\(p162\)](#))。

[「D 定数接尾子」\(p45\)](#) も参照してください。

## force\_active

説明 参照されていない関数のリンク方法をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma force_active on | off | reset`

注意 このプラグマが `on` の場合、リンカはプログラム内の決して呼ばれない関数を最終アプリケーションから外しません。このオプションはデフォルトで `off` です。

このプラグマに相当する設定パネルのオプションはありません。 `on` か否かをチェックするには、 `__option (force_active)` を使います ([「オプションのチェック」\(p162\)](#))。

fourbyteints

説明 int データ型のサイズをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma fourbyteints on | off | reset

注意 このプラグマが on の場合、int のサイズは 4 バイトです。off の場合は 2 バイトです。

このプラグマは 68K プロセッサ設定パネルの [ 4 バイト Int ] オプションに相当します。on か否かをチェックするには、\_\_option (fourbyteints を使います ( [「オプションのチェック」\(p162\)](#) ) )。

注意： できるだけこのプラグマを使わないように、設定パネルでオプションを設定してください。プラグマで設定しなければいけないときは、ファイルの先頭 ( インクルードファイルや変数、関数宣言より前 ) に置いてください。

fp\_contract

説明 特殊な浮動小数点インストラクションを使ってパフォーマンスを向上します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma fp\_contract on | off | reset

注意 このプラグマは PowerPC プログラミングでのみ有効です。

このプラグマが on の場合、コンパイラは浮動小数点演算の高速化のために FMADD、FMSUB、FN MAD 等の PowerPC インストラクションを使います。しかし on の場合、ある種の計算では予期せぬ結果が得られます。

```
register double A, B, C, D, Y, Z;  
register double T1, T2;
```

```
A = C = 2.0e23;  
B = D = 3.0e23;
```

```
Y = (A * B) - (C * D);  
printf("Y = %f\n", Y);
```

```
/* 2126770058756096187563369299968.000000 と印字 */

T1 = (A * B);
T2 = (C * D);
Z = T1 - T2;
printf("Z = %f\n", Z); /* 0.000000 と印字 */
```

上記の例では、このプラグマが `off` の場合、`Y` と `Z` は同じ値になります。

このプラグマは PPC プロセッサ設定パネルの [ FMADD と FMSUB を使用 ] オプション』に相当します。on か否かをチェックするには、`__option (fp_contract)` を使います ([「オプションのチェック」\(p162\)](#))。

## fp\_pilot\_traps

説明 Palm OS 用の浮動小数点コード生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma fp_pilot_traps on | off | reset`

注意 このプラグマは浮動小数点コードの生成をコントロールします。on の場合、コンパイラは Palm OS ライブラリルーチンを参照して浮動小数点操作を実行します。

## fullpath\_prepdump

説明 プリプロセッサ出力に含まれているファイルの完全なパスを表示します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma fullpath_prepdump on | off | reset`

注意 on の場合、`#include` 疑似命令、コメント、プリプロセッサ出力で指定されたファイルの完全なパスを表示します。off の場合、パスのファイル名の部分だけが表示されます。

このプラグマに相当する設定パネルのオプションはありません。このプラグマが on か否かをチェックするには `__option (fullpath_prepdump)` を使います ([「オプションのチェック」\(p162\)](#))。

[「line\\_prepdump」\(p111\)](#) も参照してください。

## function

説明 無視されます。Microsoft との互換性確保のために含まれています。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma function( funcname1, funcname2, ... )`

注意 無視されます。Microsoft との互換性確保のために含まれています。

## gcc\_extensions

説明 Controls the acceptance of GNU C 言語拡張の利用を制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma gcc_extensions on | off | reset`

注意 on の場合、コンパイラは C ソースコードで GNU C 拡張を受け入れます。詳細は「[GNU C 拡張](#)」(p45) を参照してください。

このプラグマが on か否かをチェックするには `__option (gcc_extensions)` を使います (「[オプションのチェック](#)」(p162))。

## global\_optimizer, optimization\_level

説明 最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma global_optimizer on | off | reset`  
`#pragma optimization_level 1 | 2 | 3 | 4 | 5`

注意 これらのプラグマは、グローバル最適化の挙動をコントロールします。グローバル最適化を on/off にするには、プラグマ `global_optimizer` を使います。グローバル最適化のレベルを選ぶには、プラグマ `optimization_level` を 1 から 5 の引数と共に使ってください。引数が高いほどグローバル最適化は強力になります。グローバル最適化がオフの場合は、コンパイラはプラグマ `optimization_level` を無視します。

コンパイラが行う最適化のレベルの詳細は各『Targeting』マニュアルを参照してください。

このプラグマはグローバル最適化設定パネルのオプションに相当します。グローバル最適化がオンか否かをチェックするには、`__option (global_optimizer)` を使います(「[オプションのチェック](#)」(p162))。

IEEEdoubles

説明 `double` 型のサイズを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma IEEEdoubles on | off | reset`

注意 このオプションは[浮動小数点モデル]ポップアップメニューの[68881]オプションと共に、`Double` の長さを指定します。次の表に、これらのオプションがどう機能するのかを示しています。

IEEE Doubles の設定	code68881 の設定	double のサイズ
on	on または off	64 ビット
off	off	80 ビット
off	on	96 ビット

このプラグマは 68K プロセッサ設定パネルの[8 バイト Doubles]オプションに相当します。on か否かをチェックするには、`__option (IEEEdoubles)` を使います(「[オプションのチェック](#)」(p162))。

注意： できるだけこのオプションを設定パネルから設定して、プラグマを使わないようにしてください。プラグマで設定しなければいけないときは、ファイルの先頭（インクルードファイルや変数、関数宣言より前）に置いてください。

ignore\_oldstyle

説明 ANS/ISO C 以前の呼び出し規約に準拠する関数宣言の認識をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------



プロトタイプ `#pragma ignore_oldstyle on | off | reset`

**注意** プラグマ `ignore_oldstyle` が `on` の場合、コンパイラは古い形式の関数宣言を無視し、関数プロトタイプ宣言を強めます。古い形式の関数宣言では、引数の型を引数リスト内で指定せず、別の行で指定していました。それは『The C Programming Language』(Kernighan、Ritchie 著、Prentice Hall 発行、初版) で使われた宣言形式です。

次のコードは古い形式の関数宣言と共にプロトタイプを定義しています。

---

```
int f(char x, short y, float z);

#pragma ignore_oldstyle on

f(x, y, z)
char x;
short y;
float z;
{
    return (int)x+y+z;
}

#pragma ignore_oldstyle reset
```

---

このプラグマは設定パネルのどのオプションとも対応していません。デフォルトではこのプラグマは `off` です。on か否かをチェックするには、`__option(ignore_oldstyle)` を使います ([「オプションのチェック」\(p162\)](#))。

**import**

**説明** シンボルのインポートをコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

<b>68K</b>	<b>PowerPC</b>	<b>NEC V800</b>	<b>Intel x86</b>	<b>MIPS</b>
------------	----------------	-----------------	------------------	-------------

プロトタイプ `#pragma import on | off | reset | list names`

**注意** このプラグマは Mac OS CFM プログラミングでのみ有効です。

このプラグマで、他のフラグメントにある変数や関数をインポートできます。CFM68K および PPC PEF 設定パネルの [シンボルをエクスポート] ポップアップメニュー、またはプラグマ `export`、`.exp` ファイルを使ってエクスポートされたシンボルをインポートするには、このプラグマを使ってください。

ある範囲内で宣言、定義されたすべての関数や変数をインポートする場合、その範囲の先頭で `#pragma import on` を使い、最後で `#pragma import off` を使います。リスト中のすべての関数や変数をインポートするには、`#pragma import list` を使います。変

数や関数を 1 つだけインポートしたい場合、その宣言の先頭に `__declspec(import)` を使ってください。

次のコードは、変数 `w` と関数 `a1()`、`b1()` をインポートするのに `#pragma import on` と `#pragma import off` を使っています。

---

```
#pragma import on
int a1(int x, double y);
double b1(int z);
int w;
#pragma import off
```

---

次のコードでは、シンボルをインポートするのに `#pragma import list` を使っています。

---

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma import list a1, b1, w
```

---

次のコードでは、シンボルをインポートするのに `__declspec(import)` を使っています。

---

```
__declspec(import) int a1(int x, double y);
__declspec(import) double b1(int z);
__declspec(import) int w;
```

---

このプラグマは設定パネルのどのオプションとも対応していません。on か否かをチェックするには、`__option (import)` を使います ([「オプションのチェック」\(p162\)](#))。

## init\_seg

説明 初期化コードを実行する順番をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `pragma init_seg( compiler | lib | user | "name " )`

注意 これは初期化コードを実行する順番を指定するプラグマです。C++ コンパイル済みモジュールの初期化コードは、静的に宣言されたオブジェクトのコンストラクタを呼び出します。C では初期化コードは生成されません。

初期化の順番は以下の通りです。

1.compiler

2.lib  
3.user

セグメント名を指定すると、初期化コードのポインタが指定されたセグメントに入れられます。この場合、初期化コードは自動的に呼び出されることはありません。明示的に呼び出すか否かを自分で決定してください。

inline\_depth

説明 インライン関数呼び出しをどの深さまで展開するのかを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma inline\_depth(n)  
                  #pragma inline\_depth(smart)

注意 インライン関数呼び出しを展開するパスの数値を設定します。n は 0 から 1024 までの整数値、または smart 規則子です。

デフォルトは smart 規則子で、パスは 4 です。小さなインライン関数のパスは 2 から 4 まですに制限されています。inline\_depth が 1 から 1024 に設定されている場合、インライン化が可能な関数がすべて展開されます。

プラグマ dont\_inline および always\_inline はこのプラグマをオーバーライドします。

inline\_intrinsics

説明 組込み関数のインラインをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma inline\_intrinsics on | off | reset

注意 on の場合、コンパイラは関数呼び出しを生成せず、直接組込み関数を生成します。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには \_\_option (inline\_intrinsics) を使います([「オプションのチェック」\(p162\)](#))。

internal

説明 モジュール外のシンボルをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma internal on | off | reset | list names`

注意 このプラグマは Mac OS CFM プログラミングでのみ有効です。

このプラグマは、特定の変数や関数がインターナルであり、かつインポートされたものではないことを指定します。ある関数が `extern` 宣言されていても、インターナル関数として呼び出す方が小さく高速なコードを生成できます。

ある範囲内で宣言、定義されたすべての関数や変数をインターナルにしたい場合、その範囲の先頭で `#pragma internal on` を使い、最後で `#pragma internal off` を使います。リスト中のすべての関数や変数をインターナルにするには、`#pragma internal list` を使います。変数や関数を 1 つだけインターナルにしたい場合、その宣言の先頭に `__declspec(internal)` を使ってください。

次のコードは、変数 `w` と関数 `a1()`、`b1()` をインターナルにするのに `#pragma internal on` と `#pragma internal off` を使っています。

```
#pragma internal on
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal off
```

次のコードでは、シンボルをインターナルにするのに `#pragma internal list` を使っています。

```
int a1(int x, double y);
double b1(int z);
int w;
#pragma internal list a1, b1, w
```

次のコードでは、シンボルをインターナルにするのに `__declspec(internal)` を使っています。

```
__declspec(internal) int a1(int x, double y);
__declspec(internal) double b1(int z);
__declspec(internal) int w;
```

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (internal)` を使います ([「オプションのチェック」\(p162\)](#))。

## interrupt

説明 割り込みルーチン用のオブジェクトコードのコンパイルをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma interrupt on|off|reset`

For Embedded PowerPC:

`#pragma interrupt [SRR DAR DSISR enable] on | off | reset`

注意 on の場合、コンパイラは関数に対して特殊なプロローグとエピローグを生成します。これによって割り込み処理が可能になります。

また、リンカが未使用コードを除去（デッドストリップ）しないように、コンパイラは割り込み関数をマークします。[「force\\_active」\(p100\)](#)を参照してください。

## interrupt\_fast

説明 割り込みルーチンのためのオブジェクトコードのコンパイルを制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma interrupt_fast on|off|reset`

注意 on の場合、コンパイラは関数に対して特殊なプロローグとエピローグを生成します。これによって割り込み処理が可能になります。

また、リンカが未使用コードを除去（デッドストリップ）しないように、コンパイラは割り込み関数をマークします。[「force\\_active」\(p100\)](#)を参照してください。

## k63d

説明 AMD K6 3D 拡張用の特殊なコード生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma k63d on | off | reset`

注意 x86 コンパイラに AMD K6 3D 用のコード生成を命じるプラグマです。特殊な 3D インストラクションを実行可能なプロセッサ上でのみ動作するコードを生成します。

このプラグマは、x86 プロセッサ設定パネルの[ 拡張命令 ] 欄の[ 3D Now! ]に相当します。

注意： この #pragma が生成するコードは Intel Pentium クラスのプロセッサと互換性はありません。

このプラグマの詳細は『Targeting Windows』マニュアルを参照してください。

### k63d\_calls

説明 AMD K6 3D 呼び出し規約の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma k63d_calls on | off | reset`

注意 x86 コンパイラに AMD K6 3D および Intel MMX 拡張を使ったコードの生成を命じるプラグマです。このプラグマは x86 プロセッサ設定パネルの[ 拡張命令 ] 欄の[ MMX ]と[ 3D Now! ] オプションに相当します。特殊なインストラクションセットを実行可能なプロセッサ上でのみ動作するコードを生成します。このプラグマが生成するコードは、モードスイッチ時に必要とするレジスタの数が普通よりも少なくなります。

このプラグマの詳細は『Targeting Windows』マニュアルを参照してください。

### lib\_export

説明 export、import、internal プラグマの認識をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma lib_export on | off | reset`

注意 このプラグマは Mac OS CFM プログラミングにのみ有効です。

このプラグマが off の場合、コンパイラは `pragma export`、`import`、`internal` を無視します。これは、コンパイラの古いバージョンとの互換性のために使用可能になってい

ます。これは、[「ANSI キーワードのみ」\(p36\)](#)で説明されている `__declspec(lib_export)` に相当します。on か否かをチェックするには、`__option (lib_export)` を使います ([「オプションのチェック」\(p162\)](#))。

このプラグマは設定パネルのどのオプションとも対応していません。

## line\_prepdump

説明 プリプロセッサ出力の `#line` 疑似命令を表示します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma line_prepdump on | off | reset`

注意 on の場合、このプラグマはプリプロセッサ出力の `#line` 疑似命令を表示します。off の場合、`#line` 疑似命令を表示しません。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (line_prepdump)` を使います ([「オプションのチェック」\(p162\)](#))。

[「fullpath\\_prepdump」\(p102\)](#) も参照してください。

## longlong

説明 `long long` 型の機能をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma longlong on | off | reset`

注意 プラグマ `longlong` が on の場合、C/C++ コンパイラで `long long` 規則子の 64 ビット整数を指定することができます。これは `long int` 型 (32 ビット整数) の 2 倍の大きさです。`long long` で -9,223,372,036,854,775,808 から 9,223,372,036,854,775,807 までの値を指定できます。unsigned `long long` では 0 から 18,446,744,073,709,551,615 までの値を指定できます。

このプラグマは設定パネルのどのオプションとも対応していません。on か否かをチェックするには、`__option (longlong)` を使います ([「オプションのチェック」\(p162\)](#))。

## longlong\_enums

説明 `long long` 型のサイズの `enum` 型をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma longlong_enums on | off | reset`

注意 このプラグマで `longlong` 整数を保持できる大きさの `enum` 定数を使うことができます。プラグマ `enumsalwaysint` が `on` の場合でもそれを無視します。詳細は「[enumsalwaysint](#)」(p93) を参照してください。

コンパイラが `enum` 型をどのように処理するかについて詳しくは「[enum は常に Int 型](#)」(p31) を参照してください。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (longlong_enums)` を使います(「[オプションのチェック](#)」(p162))。デフォルトではこのオプションは `on` です。

## longlong\_prepval

説明 プリプロセッサに `long` 型の式を `long long` 型として扱うか否かを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma longlong_prepval on | off | reset`

注意 このプラグマが `on` の場合、C/C++ プリプロセッサは `long` 型の式を `long long` 型として扱います。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (longlong_prepval)` を使います(「[オプションのチェック](#)」(p162))。デフォルトではこのオプションは `on` です。

## macsbug, oldstyle\_symbols

説明 MacsBug 用のデバッガデータを生成します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------



プロトタイプ     `#pragma macsbug on | off | reset`  
                  `#pragma oldstyle_symbols on | off | reset`

注意 このプラグマは 68K Mac OS プログラミングでのみ有効です。

これらのプラグマで、コンパイラがどのような MacsBug シンボルを作るのかを選択できます。CodeWarrior デバッガを含む多くのデバッガは、関数や変数の名前を表示するのに MacsBug シンボルを使います。プラグマ `macsbug` で、MacsBug シンボルの生成を `on/off` にします。プラグマ `oldstyle_symbols` で、どのタイプのシンボルを生成するのかを選択します。次の表に、これらのプラグマの働きを示します。

目的	使用するプラグマ
Macsbug シンボルを作らない	<code>#pragma macsbug on</code>
古い形式の Macsbug シンボルを作る	<code>#pragma macsbug on</code> <code>#pragma oldstyle_symbols on</code>
新しい形式の Macsbug シンボルを作る	<code>#pragma macsbug on</code> <code>#pragma oldstyle_symbols off</code>

これらのプラグマは 68K プロセッサ設定パネルの [ MacsBug シンボル ] ポップアップメニューの [ 旧スタイル ] に相当します。プラグマ `macsbug` が `on` か否かをチェックするには、`__option (macsbug)` を使います。プラグマ `oldstyle_symbols` が `on` か否かをチェックするには、`__option (oldstyle_symbols)` を使います([「オプションのチェック」\(p162\)](#))。

mark

説明 IDE エディタウィンドウの関数ポップアップメニューにアイテムを追加します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     `#pragma mark itemName`

注意 このプラグマで、ソースファイルの関数ポップアップメニューに `itemName` を追加することができます。そのファイルを CodeWarrior エディタで開き、関数ポップアップメニューでそのアイテムを選ぶと、このプラグマの場所が表示されます。このプラグマが関数定義の内部にあるときは、このアイテムは関数ポップアップメニューに現われません。

`itemName` が `--` で始まる場合、メニューセパレータが IDE の関数ポップアップメニューに現れます。

`#pragma mark --`

このプラグマに相当する設定パネルのオプションはありません。

## message

説明 ユーザーへテキストメッセージを発行します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma message("text")`

注意 このプラグマは、ユーザーへのメッセージやテキストの発行をコンパイラに命じます。CodeWarrior IDE 上で動作しているとき、メッセージは「エラーと警告」ウィンドウに表示されます。

## microsoft\_exceptions

説明 Microsoft C++ 例外処理を行います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma microsoft_exceptions on | off | reset`

注意 x86 コンパイラに Microsoft C++ 例外処理コードと互換性のある例外処理コードの生成を命じるプラグマです。

このプラグマが on か否かをチェックするには、`__option (microsoft_exceptions)` を使います ([「オプションのチェック」\(p162\)](#))。

## microsoft\_RTTI

説明 Microsoft C++ ランタイムタイプ情報を生成します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma microsoft_RTTI on | off | reset`

注意 x86 コンパイラに Microsoft C++ と互換性のあるランタイムタイプ情報を生成させるプラグマです。

プラグマが on か否かをチェックするには、`__option (microsoft_RTTI)` を使います ([「オプションのチェック」\(p162\)](#))。

mmx

説明 Intel MMX 拡張を使った特殊なコードを生成します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma mmx on | off | reset

注意 このプラグマは x86 コンパイラに Intel MMX 拡張を使ったコードを生成させます。コンパイラが生成するコードは 50 以上の MMX インストラクションを実行可能なプロセッサ上でのみ動作します。

このプラグマは x86 プロセッサ設定パネルの [ 拡張命令 ] 欄の [ MMX ] オプションに相当します。on か否かをチェックするには、\_\_option (mmx) を使います (「[オプションのチェック](#)」(p162))。

詳細は『Targeting Windows』マニュアルを参照してください。

mmx\_call

説明 MMX 呼び出し規約の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ #pragma mmx\_call on | off | reset

注意 プラグマが on の場合、コンパイラは MMX 呼び出し規約を使用します。

このプラグマが on か否かをチェックするには、\_\_option (mmx\_call) を使います (「[オプションのチェック](#)」(p162))。

詳細は『Targeting Windows』マニュアルを参照してください。

mpwc

説明 Apple 社の MPW C 呼び出し規約の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma mpwc on | off | reset

注意 このプラグマは Mac OS 68K プロセッサ上でのみ有効です。

プラグマ mpwc が on の場合、コンパイラは MPW C の呼びだし規約との互換性を保つために以下のことをします。

2 バイトより小さい整数引数を符号拡張した long integer として渡します。コンパイラは次の宣言を、

```
int MPWfunc ( char a, short b, int c, long d, char *e );
```

以下のように変換します。

```
long MPWfunc( long a, long b, long c, long d, char *e );
```

浮動小数点引数をすべて long Double として渡します。コンパイラは次の宣言を、

```
void MPWfunc( float a, double b, long double c );
```

以下のように変換します。

```
void MPWfunc( long double a, long double b, long double c );
```

(pragma pointers\_in\_D0 が off でも) すべてのポインタ値を D0 に入れて戻します。

1 バイト、2 バイト、4 バイト構造体をすべて D0 に入れて戻します。

[ 68881 ] オプションがオンの場合、浮動小数点値はすべて FP0 に入れて戻します。

このプラグマは 68K プロセッサ設定パネルの [ MPW C 呼び出し規約 ] オプションに相当します。on か否かをチェックするには、\_\_option (mpwc) を使います ([「オプションのチェック」\(p162\)](#))。

mpwc\_newline

説明 Apple 社の MPW C 規約の改行文字を使います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma mpwc_newline on | off | reset`

**注意** プラグマ `mpwc_newline` が `on` の場合、コンパイラは、`\n`、`\r` 文字に対して MPW の規約を使います。`off` の場合、コンパイラはこれらの文字に対して、Metrowerks C/C++ の規約を使います。

MPW では、`\n` はキャリッジリターン (0x0D) で、`\r` はラインフィード (0x0A) です。Metrowerks C/C++ ではそれらは反対で、`\n` がラインフィードで、`\r` がキャリッジリターンです。

このプラグマが `on` の場合、ANSI C/C++ ライブラリがこのプラグマが `on` でコンパイルされたものかどうか確認してください。これらのライブラリは `NL` と記されています。68K バージョンでは、`MSL C.68K (NL_2i).Lib` となります。PowerPC バージョンでは、`MSL C.PPC(NL).Lib` となります。

このプラグマを `on` にして標準 ANSI C/C++ ライブラリを使うと、`\n` と `\r` を正しく読み書きできません。例えば、`\n` と印字してもその行の先頭に来るだけで次の行にはなりません。

このプラグマは C/C++ 言語設定パネルの [ CR の代わりに NL を利用 ] オプションに相当します。`on` か否かをチェックするには、`__option (mpwc_newline)` を使います ([「オプションのチェック」\(p162\)](#))。

## mpwc\_relax

**説明** `char*` と `unsigned char*` 型の互換性をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma mpwc_relax on | off | reset`

**注意** このプラグマが `on` の場合、コンパイラは `char*` と `unsigned char*` を同じ型とみなします。このオプションは ANSI C 規格以前に書かれたコードに対して特に有効です。この古いコードは、これらの型を相互変換のためにしばしば使います。

このプラグマは C++ ソースコードには効果はありません。

このプラグマは関数のポインタチェックの緩和に使用されます。

---

```
#pragma mpwc_relax on
extern void f(char *);
extern void(*fp1)(void *) = &f;           // エラーだが許容される
extern void(*fp2)(unsigned char *) = &f;  // エラーだが許容される
```

---

このプラグマは [C/C++ 言語設定パネル](#) の [ ポインタタイプルールを緩める ] オプションに相当します。on か否かをチェックするには、`__option (mpwc_relax)` を使います([「オプションのチェック」\(p162\)](#))。

## no\_register\_coloring

説明 複数の変数の値を格納するレジスタの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma no_register_coloring on | off | reset`

注意 プラグマ `no_register_coloring` が `off` の場合、コンパイラはレジスタカラーリングを行います。この最適化によって、複数の変数が 1 つのレジスタを共有します。複数の変数を同時に使用しない場合、異なる変数またはパラメータを 1 つのレジスタに割り当てます。以下の例では、コンパイラは `i` と `j` を同じレジスタに割り当てます。

```
short i;
int j;

for (i=0; i<100; i++) {    MyFunc(i);    }
for (j=0; j<1000; j++) {    OurFunc(j);    }
```

`i` と `j` を同時に使用する場合、関数に以下のような行を入れると、コンパイラは別々のレジスタを割り当てます。

```
int k = i + j;
```

レジスタカラーリングを `on` にしてデバッグを行うと、レジスタを共有している変数にエラーがあるように見えることがあります。上記の例では、`i` と `j` が常に同じ値になってしまいます。`i` が変わると `j` も同じに変わります。デバッグ中のこの混乱を避けるためには、レジスタカラーリングを `off` にするか、変数を `volatile` 宣言してください。

このプラグマはグローバル最適化設定パネルの [ 大域レジスタ割り当て ] に相当します。on か否かをチェックするには、`__option (no_register_coloring)` を使います([「オプションのチェック」\(p162\)](#))。デフォルトは `off` です。

注意： PowerPC Mac OS でレジスタカラーリングをオフにするには、`#pragma global_optimizer off` を使います。詳しくは [「global\\_optimizer, optimization\\_level」\(p103\)](#) を参照してください。

[「register\\_coloring」\(p132\)](#) を参照してください。

## no\_static\_dtors

説明 C++ の静的デストラクタの生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma no_static_dtors on | off | reset`

注意 プラグマが `on` の場合、コンパイラは静的またはグローバルオブジェクトデストラクタのためのオブジェクトコードを生成しません。オブジェクトコードのサイズを減らすためです。

このプラグマに相当する設定パネルのオプションはありません。デフォルトは `off` です。  
`on` か否かをチェックするには、`__option (no_static_dtors)` を使います([「オプションのチェック」\(p162\)](#))。

## once

説明 1 つのソースファイルは 1 度だけインクルードされるヘッダファイルを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma once [ on | off ]`

注意 このプラグマで、コンパイラがソースファイルに対して 1 度だけヘッダファイルをインクルードするように指示することができます。このプラグマは、プリコンパイル済みのヘッダファイルに対して特に有効です。

このプラグマには、`#pragma once` と `#pragma once on` という 2 つのバージョンがあります。`#pragma once` は、ヘッダファイル内で、そのヘッダファイルがソースファイルに対して 1 度だけインクルードされるように指示するために使います。`#pragma once on` は、ヘッダファイル内またはソースファイル内で、すべてのファイルがソースファイルに対して 1 度だけインクルードされるように指示するために使います。

このプラグマは設定パネルのどのオプションとも対応していません。デフォルトではこのプラグマは `off` です。

## only\_std\_keywords

説明 ANSI/ISO キーワードの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma only_std_keywords on | off | reset`

注意 C/C++ コンパイラは、追加の予約済みキーワードを認識します。ANSI 規格に厳密に準拠したコードを書く場合、プラグマ `only_std_keywords` を `on` にしてください。詳しくは、[「ANSI キーワードのみ」\(p36\)](#) を参照してください。

このプラグマは C/C++ 言語設定パネルの [ ANSI キーワードのみ ] オプションに相当します。`on` か否かをチェックするには、`__option (only_std_keywords)` を使います([「オプションのチェック」\(p162\)](#))。

### `opt_common_subs`

説明 共通サブ式の最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_common_subs on | off | reset`

注意 `on` の場合、コンパイラは同様の冗長式を 1 つの式で置換します。例えば、関数内の 2 つのステートメントで以下の式を使っている場合、

`a * b * c + 10`

コンパイラは式を 1 度だけ計算するオブジェクトコードを生成して、その結果を 2 つの式に適用します。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (opt_common_subs)` を使います([「オプションのチェック」\(p162\)](#))。

### `opt_dead_assignments`

説明 デッドストア最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_dead_assignments on | off | reset`

注意 `on` の場合、変数が再度代入される前に使用されていなければ、コンパイラはその変数への代入を除去します。



このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (opt_dead_assignments)` を使います ([「オプションのチェック」\(p162\)](#))。

## opt\_dead\_code

説明 デッドコード最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_dead_code on | off | reset`

注意 on の場合、コンパイラは他のステートメントから参照されないステートメントや、実行されないステートメントを削除します。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (opt_dead_code)` を使います ([「オプションのチェック」\(p162\)](#))。

## opt\_lifetimes

説明 生存期間分析最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_lifetimes on | off | reset`

注意 on の場合、変数が同一ステートメントで使用されていなければ、コンパイラは同一ルーチンにある別々の変数に対して同じプロセッサレジスタを使います。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (opt_lifetimes)` を使います ([「オプションのチェック」\(p162\)](#))。

## opt\_loop\_invariants

説明 ループ内最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_loop_invariants on | off | reset`

注意 `on` の場合、コンパイラはループ内で変化しない演算を外に移動してループの速度を上げます。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (opt_loop_invariants)` を使います ([「オプションのチェック」\(p162\)](#))。

`opt_propagation`

説明 コピーと定数のプロパゲーションの最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_propagation on | off | reset`

注意 `on` の場合、コンパイラは1つの変数の複数の出現箇所を1つにまとめます。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (opt_propagation)` を使います ([「オプションのチェック」\(p162\)](#))。

`opt_strength_reduction`

説明 強度のコード簡約化最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma opt_strength_reduction on | off | reset`

注意 `on` の場合、コンパイラはループ内部の乗算命令を加算命令に変換してループの速度を上げます。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (opt_strength_reduction)` を使います ([「オプションのチェック」\(p162\)](#))。

`opt_unroll_loops`

説明 ループのアンローリング最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     `#pragma opt_unroll_loops on | off | reset`

注意 on の場合、コンパイラは速度を改善するために、ループ内のステートメントを展開して複数配置します。

このプラグマに相当する設定パネルのオプションはありません。on が否かをチェックするには、`__option (opt_unroll_loops)` を使います([「オプションのチェック」\(p162\)](#))。

opt\_vectorize\_loops

説明 ループのベクトル化による最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     `#pragma opt_vectorize_loops on | off | reset`

注意 on の場合、ループのパフォーマンスが上がります。

注意： ループのベクトル化と、PowerPC AltiVec ベクトルインストラクションを混同しないでください。ループのベクトル化は、パフォーマンスを改善するためにループ内のインストラクションを並べ替えます。PowerPC AltiVec ベクトルインストラクションは、ベクトルを操作するための特殊なインストラクションで、一部の PowerPC プロセッサでのみ利用可能です。AltiVec については [「altivec\\_codegen」\(p75\)](#)、[「altivec\\_model」\(p76\)](#)、[「altivec\\_vrsave」\(p76\)](#) を参照してください。

このプラグマに相当する設定パネルのオプションはありません。on が否かをチェックするには、`__option (opt_vectorize_loops)` を使います ([「オプションのチェック」\(p162\)](#))。

optimization\_level

`global_optimizer (「global\_optimizer, optimization\_level」\(p103\))` を参照してください。

optimize\_for\_size

説明 オブジェクトコードのサイズを減らす最適化をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma optimize_for_size on | off | reset`

注意 最適化の基準をサイズに置くか速度に置くかを選択します。on の場合、コンパイラはサイズの小さいオブジェクトコードを生成しますが、速度は遅くなります。off の場合、オブジェクトコードの速度は早くなりますがサイズは大きくなります。

on の場合、コンパイラは inline 疑似命令を無視し、inline 宣言された関数を呼び出す関数を生成します。

このプラグマはグローバル最適化設定パネルの [ コードサイズの縮小か ] オプションに相当します。on か否かをチェックするには、`__option (optimize_for_size)` を使います (「[オプションのチェック](#)」(p162))。

oldstyle\_symbols

このプラグマの詳細は「[macsbug, oldstyle\\_symbols](#)」(p112) を参照してください。

pack

説明 データ構造体のアライメントをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma pack( [n | push, n | pop] )`

注意 データ構造体のパック配置を設定します。この設定は、次にプラグマ pack で設定を変えるまで、このプラグマ以降に宣言されたすべてのデータ構造体に対して有効です。

プラグマ	働き
<code>#pragma pack(n)</code>	配置モジュールを <i>n</i> にする。 <i>n</i> は 1、2、4、8、16 のいずれかです。MIPS コンパイラでは <i>n</i> が 0 の場合、構造体配置をデフォルトの設定に戻します。
<code>#pragma pack(push, n)</code>	カレントの配置モジュールをスタック上にプッシュし <i>n</i> にします。 <i>n</i> は 1、2、4、8、16 のいずれか。ある宣言 (1 つまたは複数) に対して、特定のモジュールが必要だが、デフォルトの設定を変えたくない場合、push と pop を使います。MIPS コンパイラはこれをサポートしません。

プラグマ	働き
#pragma pack(pop)	pushされた配置モジュールをスタックからpopします。 MIPS コンパイラはこれをサポートしません。
#pragma pack()	x86 コンパイラは配置モジュールを x86 CodeGen 設定パ ネルの設定にリセットします。MIPS コンパイラは構造 体配置をデフォルトの設定に戻します。

このプラグマは x86 プロセッサ設定パネルの[ バイトアライメント ]ポップアップメニュー  
に相当します。

parameter

説明 パラメータを渡すレジスタを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma parameter *return-reg func-name(param-regs)*

注意 このプラグマは 68K プログラミングでのみ有効です。

コンパイラは、関数 *func-name* に対して、パラメータを *param-regs* で指定されたレジ  
スタ(スタックではない)に渡し、戻り値を *return-reg* で指定したレジスタにいて戻  
します。*return-reg* と *param-regs* はオプションです。

以下に例を示します。

```
#pragma parameter __D0 Gestalt(__D0, __A1)
#pragma parameter __A0 GetZone
#pragma parameter HLock(__A0)
```

関数を定義するとき、パラメータリストでレジスタを指定する必要があります。詳細は各  
『Targeting』マニュアルを参照してください。

このプラグマは設定パネルのどのオプションとも対応していません。

pcrelstrings

説明 プログラムカウンタからの文字列の参照と保存をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma pcrelstrings on | off | reset`

**注意** このプラグマが `on` の場合、コンパイラは文字列定数をコードセグメントのローカルスコープにストアし、その文字列を `PC` 相対命令でアクセスします。`off` の場合、コンパイラはすべての文字列定数をグローバルデータセグメントにストアします。このオプションの設定にかかわらず、コンパイラは、グローバルスコープで使われる文字列定数をグローバルデータセグメントにストアします。

---

```
#pragma pcrelstrings on
int foo(char *);

int x = f("Hello"); // "Hello" はグローバルデータセグメントに
                    // 割り当てられる

int bar()
{
    return f("World"); // "World" はコードセグメントに
                       // 割り当てられる (pc 相対)
}
```

---

C++ 初期化コード内の文字列は、常にグローバルデータセグメントに割り当てられます。

---

**注意：** プラグマ `pool_strings` が `on` の場合、コンパイラはプラグマ `pcrelstrings` の設定を無視します。

---

このプラグマは 68K プロセッサ設定パネルの [ `PC- 相対文字列` ] オプションに相当します。`on` か否かをチェックするには、`__option (pcrelstrings)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトではこのオプションは `off` です。

## peephole

**説明** ピープホール最適化をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma peephole on | off | reset`

**注意** このプラグマが `on` の場合、コンパイラはピープホール最適化(比較命令と分岐順序を改善する局所的な小規模の最適化)を行います。

このプラグマは PPC プロセッサ設定パネルの [ `ピープホール最適化` ] オプションに相当します。`on` か否かをチェックするには、`__option (peephole)` を使います ([「オプションのチェック」\(p162\)](#))。

## ppc\_unroll\_factor\_limit

説明「アンロール」されたループへ置くためにループイテレーションの数を制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma ppc_unroll_factor_limit number`

注意「アンロール」されたループへ置くループ本体のコピーの最大数を指定します。ループのアンローリング最適化はプラグマ [opt\\_unroll\\_loops](#) で制御します。

*number* のデフォルト値は 10 です。

## ppc\_unroll\_instructions\_limit

説明「アンロール」されたループで許可されたインストラクションの数を制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma ppc_unroll_instructions_limit number`

注意「アンロール」されたループへ置くインストラクションの最大数を指定します。ループのアンローリング最適化はプラグマ [opt\\_unroll\\_loops](#) で制御します。

*number* のデフォルト値は 100 です。

## ppc\_unroll\_speculative

説明 ランタイム時のループの「アンロール」を制御します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma ppc_unroll_speculative on | off | reset`

注意 このプラグマがオンの場合、コンパイラはループイテレーションの数をランタイム時に計算し（コンパイル時に計算された定数値ではなく）、ループをアンロールする回数を判断します。

以下の条件を満たすときにのみ、この最適化は行われます。

- ループアンローリングが on である
- ループイテレータが 32 ビット値である (int, long, unsigned int, unsigned long)
- ループ本体に条件ステートメントがない

このプラグマが on の場合、ループアンローリング要素は 2 の 2 乗、プラグマ [ppc\\_unroll\\_factor\\_limit](#) で指定された値よりも少ない、または等しいです。

ループアンローリング最適化はプラグマ [opt\\_unroll\\_loops](#) で制御します。ループアンローリングが on の場合、このプラグマはデフォルトで on です。on か否かをチェックするには、\_\_option (ppc\_unroll\_speculative) を使います (「[オプションのチェック](#)」(p162))

pointers\_in\_A0, pointers\_in\_D0

説明 使用する呼び出し規約を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma pointers\_in\_A0  
                    #pragma pointers\_in\_D0

注意 このプラグマは Mac OS 68K プロセッサ上でのみ利用できます。

これらのプラグマで、MPW や Macintosh Toolbox ルーチンとの Metrowerks C/C++ ルーチンの 2 種類の呼び出し規約を選べます。MPW や Macintosh Toolbox の呼び出し規約では、関数はポインタをレジスタ D0 に入れて戻します。Metrowerks C/C++ の呼び出し規約では、関数はポインタをレジスタ A0 に入れて戻します。

Macintosh Toolbox や MPW でコンパイルされたライブラリの関数を宣言するときは、プラグマ pointers\_in\_D0 を使います。それらの関数を宣言した後にプラグマ pointers\_in\_A0 を使って Metrowerks C/C++ 関数を呼び出せるようにします。

[例 6.3](#) では、Sound.h 内の Toolbox 関数はポインタを D0 に入れて戻し、Myheader.h 内で定義されたユーザー定義関数はポインタを A0 に入れて戻します。

例 6.3           #pragma pointers\_in\_A0 と #pragma pointers\_in\_D0 の使用

```
#pragma pointers_in_D0 // Toolbox コール用設定
#include <Sound.h>
#pragma pointers_in_A0 // 独自のルーチン用設定
#include "Myheader.h"
```



プラグマ `pointers_in_A0` と `pointers_in_D0` は、`d0_pointers` とほとんど同じ意味で、これらは下位互換性を確保するため使用可能になっています。プラグマ `pointers_in_A0` は `#pragma d0_pointers off` に相当します。プラグマ `pointers_in_D0` は `#pragma d0_pointers on` に相当します。`reset` 引数があるので、新規のコードにはプラグマ `d0_pointers` の使用を推奨します。詳しくは、[「d0\\_pointers」\(p84\)](#) を参照してください。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (d0_pointers)` を使います ([「オプションのチェック」\(p162\)](#))。

## pool\_data

説明 データの保存方法を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma pool_data on | off | reset`

注意 このプラグマはエンベデッド PowerPC プログラミングでのみ有効です。

`on` の場合、コンパイラはプールされたデータを最適化します。プラグマを適用する関数よりも先にプラグマを使わなくてはなりません。

このプラグマは PPC Processor 設定パネルの [ Pool Data ] オプションに相当します。`on` か否かをチェックするには、`__option (pool_data)` を使います ([「オプションのチェック」\(p162\)](#))。

## pool\_strings

説明 文字列の保存方法を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma pool_strings on | off | reset`

注意 [C/C++ 言語設定パネル](#)の [ 文字列定数を一ヶ所にまとめる ] オプションがオンの場合、コンパイラはすべての文字列定数を集めて 1 つのデータオブジェクトにするので、それらに対して 1 つの TOC エントリが必要になります。このプラグマが `off` の場合、コンパイラはそれぞれの文字列定数に対して独立したデータオブジェクトと TOC エントリを作ります。プラグマが `on` の場合、プログラム内の TOC エントリの数は減りますが、文字列のアドレスを収めるのにあまり効率の良い手法をとらないので、プログラムサイズが大きくなります。

このプラグマは、プログラムが大きく、多くの文字列定数がある場合や Metrowerks Profiler を使う場合に特に有効です。

注意： プラグマ `pool_strings` が `on` の場合、コンパイラはプラグマ `pcrelstrings` の設定を無視します。

このプラグマは C/C++ 言語設定パネルの [ 文字列定数を一ヶ所にまとめる ] オプションに相当します。 `on` か否かをチェックするには、 `__option (pool_strings)` を使います (「[オプションのチェック](#)」(p162))。

pop, push

説明 プラグマの設定を保存、復帰します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     `#pragma push`  
                  `#pragma pop`

注意 プラグマ `push` は、現在のすべてのプラグマ設定を保存します。プラグマ `pop` は、最後にプラグマ `push` が呼ばれたときのすべてのプラグマ設定をリストアします。[例 6.4](#) に例を示します。

例 6.4     push と pop の例

```
#pragma far_data on
#pragma pointers_in_A0
#pragma push    // すべてのコンパイラオプションを push する
#pragma far_data off
#pragma pointers_in_D0
    // "far_data" と "pointers_in_A0" を pop してリストアする
#pragma pop
```

これらのプラグマは、Metrowerks C/C++ で MacApp を使うために用意されました。新たにコードを書くときにプラグマオプションをオリジナルの値に戻すには、「[プラグマの文法](#)」(p72)にある `reset` 引数を使います。

precompile\_target

説明 プリコンパイルヘッダファイルの名前を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma precompile_target filename`

注意 このプラグマは、*Filename* をプリコンパイルヘッダファイルとして指定します。*Filename* を指定しないとき、コンパイラはプリコンパイルヘッダファイル名にソースファイルと同じ名前を与えます。

*Filename* は単にファイル名でも絶対パス名でも構いません。*Filename* が単にファイル名するとき、コンパイラはソースファイルと同じフォルダにファイルをセーブします。*Filename* がパス名するとき、コンパイラは指定されたフォルダにファイルを保存します。

[例 6.5](#) は、MacHeaders プリコンパイルヘッダのソースファイルからのサンプルソースコードです。定義済みシンボル `__cplusplus` と `powerc` およびプラグマ `precompile_target` を使うことにより、コンパイラは同じソースコードから、C、C++、680x0、PowerPC 用の異なるプリコンパイルヘッダファイルを作ることができます。

例 6.5 `#pragma precompile_target filename` の使用

```
#ifdef __cplusplus
    #ifdef powerc
        #pragma precompile_target "MacHeadersPPC++"
    #else
        #pragma precompile_target "MacHeaders68K++"
    #endif
#else
    #ifdef powerc
        #pragma precompile_target "MacHeadersPPC"
    #else
        #pragma precompile_target "MacHeaders68K"
    #endif
#endif
```

profile

説明 CodeWarrior Profiler 用のオブジェクトコードを生成します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma profile on | off | reset`

注意 このプラグマは Mac OS プログラミングでのみ有効です。

このプラグマが `on` の場合、コンパイラはそれぞれの関数に対して、Metrowerks Profiler 用の情報を含むコードを生成します。詳しくは『Profiler Manual』を参照してください。

このプラグマは 68K プロセッサ設定パネルの [ プロファイラ情報を生成 ] オプションと、PPC プロセッサ設定パネルの [ プロファイラ情報を生成 ] オプションに相当します。`on` か否かをチェックするには、`__option (profile)` を使います ([「オプションのチェック」\(p162\)](#))。

`readonly_strings`

説明 文字列の保存方法をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma readonly_strings on | off | reset`

注意 このオプションで、文字列定数がどこにストアされるのかが決まります。このプラグマが `off` の場合、コンパイラは文字列定数をデータセクションにストアします。`on` の場合、コンパイラは文字列定数をコードセクションにストアします。

注意： ランタイム時に他のオブジェクトのアドレスに初期化されない変数は、常にコードセクション（クラス RO）に置かれます。`const` 宣言された C/C++ 変数もこれに含まれます。

このプラグマは PPC プロセッサ、MIPS、V800 プロセッサ設定パネルの [ 文字列を読み取り専用にする ] オプションに相当します。`on` か否かをチェックするには、`#if __option (readonly_strings)` を使います ([「オプションのチェック」\(p162\)](#))。

`register_coloring`

説明 レジスタカラーリングの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma register_coloring on | off | reset`

注意 `on` の場合、コンパイラは同時に利用されない複数の変数を同一のレジスタに格納し、パフォーマンスを向上させます。

ヒント: プログラムをデバッグするときは off にしてください。

このプラグマは x86 プロセッサ設定パネルの [ SYM 形式 ] オプションに相当します。on か否かをチェックするには、`__option (register_coloring)` を使います ([「オプションのチェック」\(p162\)](#))。

[「no\\_register\\_coloring」\(p118\)](#) を参照してください。

## require\_prototypes

説明 関数のプロトタイプを必要とするか否かを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma require_prototypes on | off | reset`

注意 プラグマ `require_prototypes` が on の場合は、コンパイラはプロトタイプのない関数に対してエラーを出します。このプラグマは、関数を定義する以前に使うときに起きるエラーを防ぐのに役立ちます。

このプラグマは C/C++ 言語設定パネルの [ 関数プロトタイプが必要 ] オプションに相当します。on か否かをチェックするには、`__option (require_prototypes)` を使います ([「オプションのチェック」\(p162\)](#))。

## 実行時型情報 (RTTI)

説明 RTTI の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma RTTI on | off | reset`

注意 プラグマ `RTTI` が on の場合、`dynamic_cast` や `typeid` などの RTTI (Run-Time Type Information) 機能が使えます。他の RTTI 機能は [ RTTI 有効 ] オプションがオフでも使用可能です。`*type_info::before(const type_info&)` は実装されていません。

このプラグマは C/C++ 言語設定パネルの [ RTTI 有効 ] オプションに相当します。on か否かをチェックするには、`__option (RTTI)` を使います ([「オプションのチェック」\(p162\)](#))。

## schedule

説明 インストラクションスケジュール最適化の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma schedule once | twice | altivec`

注意 コンパイラがオブジェクトコードをインストラクションスケジューラーへ渡す回数を指定します。

高度に最適化された C コードでは、ループは手動でアンロールされます。特に `register` 指定子を使う関数では、スケジューラーを 2 回実行するよりも、1 回だけ実行する方が良い結果を出すことがあります。

スケジューラーを 2 回実行する場合、レジスタカラーリングの前と後に実行されます。1 回実行する場合、レジスタカラーリングの後だけです。

このプラグマのデフォルト値は `twice` です。[「no\\_register\\_coloring」\(p118\)](#) を参照してください。

## scheduling

説明 インストラクションスケジュールの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma scheduling 601 | 603 | 604 | on | off | reset`

注意 このプラグマで、コンパイラが高速化のためにどのように命令を並べ変えるかを指定できません。メモリロードなどの命令は 1 プロセッササイクルより時間がかかります。ロードとロードされたデータを使うまでの間のそれと無関係な命令を移動させることによって、コンパイラは実行時のサイクルを節約することができます。

PowerPC 用には、601、603、604 を選べます。`on` を使うと、コンパイラは 601 スケジューリングを実行します。

コードをデバッグするときは、このプラグマを `off` にしてください。コードから生成された命令が並べ替えられてしまうので、デバッガがソースコードと命令の対応関係を把握できないためです。

## section

説明 オブジェクトコードの並べ換えを行います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma section [ objecttype | permission ] [ iname ] [ uname ]  
[ data_mode=datamode ] [ code_mode=codemode ]`

注意 このプラグマはエンベデッド PowerPC プログラミングでのみ有効です。

コンパイルしたオブジェクトコードを定義済みのセクションや自分で定義したセクションに並べ換えるプラグマです。

[引数](#)

[セクションアクセス権](#)

[定義済みセクションとデフォルトセクション](#)

[#pragma section のフォーム](#)

[個々のオブジェクトを特定のセクションへ入れる](#)

[#pragma push、#pragma pop と #pragma section を使う](#)

## 引数

オプションのパラメータ *objecttype* でオブジェクトデータをストアする場所を指定します。以下の値のいずれか（複数も可）になります。

*code\_type* : 実行可能オブジェクトコード

*data\_type* : PowerPC EABI Project 設定パネルで指定した small データのしきい値よりも大きいサイズの定数以外のデータ

*sdata\_type* : PowerPC EABI Project 設定パネルで指定した small データのしきい値以下のサイズの定数以外のデータ

*const\_type* : PowerPC EABI Project 設定パネルで指定した small const データのしきい値よりも大きいサイズの定数データ

*sconst\_type* : PowerPC EABI Project 設定パネルで指定した small const データのしきい値以下のサイズの定数データ

*all\_types* : 全データ

複数のオブジェクトタイプを指定するには、クォートなしでスペースで区切ります。

CodeWarrior C/C++ は独自のデータ（例外や静的イニシャライザオブジェクトなど）を生成しますが、これらは `#pragma section` の影響を受けません。

注意： CodeWarrior C/C++ は PowerPC EABI プロセッサ設定パネルの [ Make Strings Read Only ] オプションの初期設定を使ってキャラクタ文字列を区別します。[ Make Strings Read Only ] がオンの場合、キャラクタ文字列は `const_type` のデータ型と同じセクションにストアされます。オフの場合、`data_type` のデータ型と同じセクションにストアされません。

オプションのパラメータ `permission` でファイルアクセス権を指定します。以下の値のいずれか（複数も可）になります。

R：リードオンリー

W：書き込み可

X：実行可

アクセス権については「[セクションアクセス権](#)」(p137) を参照してください。複数のアクセス権を指定するには、クォートなし、スペースなしで指定します。

オプションのパラメータ `iname` は、コンパイラが初期化したオブジェクトを保存したセクションの名前を指定する引用名です。変数は定義されたときに初期化され、関数とキャラクタ文字列は初期化されたオブジェクトの例です。パラメータ `iname` の形式は `.abs.xxxxxxxx` です。`xxxxxxx` は、セクションのアドレスを指定する 8 桁の 16 進法数値です。

オプションのパラメータ `uname` はコンパイラが初期化していないオブジェクトを保存したセクションの名前を指定する引用名です。データオブジェクトを持つセクションにはこのパラメータが必要です。パラメータ `uname` はユニーク、または前述の `iname` か `uname` セクションと同じ名前になります。`uname` セクションと `iname` セクションが同じ場合、未初期化データは初期化済みオブジェクトと同じセクションに保存されます。

特殊な `uname COMM` は、初期化済みデータを `common` セクションに保存します。リンカはすべての `common` セクションデータを `.bss` セクションに入れます。PowerPC EABI Processor 設定パネルの [ Use Common Section ] オプションがオンの場合、`COMM` がデフォルトの `uname` で、`.data` セクションを指定します。[ Use Common Section ] オプションがオフの場合、`COMM` がデフォルトの `uname` で `.data` セクションを指定します。

オプションのパラメータ `uname` では一定ではありません。例えば、大部分の未初期化データを `.bss` セクションに入れ、特定の変数を `COMM` セクションに入りたい場合などです。[例 6.6](#) は特定の未初期化変数を `COMM` セクションに保存する例です。

#### 例 6.6 COMM セクションに未初期化データを保存

```
// Use Common Section オプションがオン
#pragma push // 現在の状態を保存
#pragma section ".data" "COMM"
int foo;
int bar;
#pragma pop // 前の状態をリストア
```



セクション名にオブジェクトタイプ、データモード、コードモードを使うことはできません。PowerPC EABI 設定パネルで指定した独自に定義済みセクション名も使えません。

オプションのパラメータ `data_mode=datamode` は、セクションのデータオブジェクトを参照するときのアドレッシングモードを指定します。

`datamode` で指定可能なアドレッシングモードを以下に示します。

`near_abs` : オブジェクトは RAM の最初の 16 ビット以内にある

`far_abs` : オブジェクトは RAM の最初の 32 ビット以内にある

`sda_rel` : オブジェクトはリンクが定義した small データベースアドレスの 32K 以内にある

`sda_rel` アドレッシングモードを使えるのは `.sdata`、`.sbss`、`.sdata2`、`.sbss2`、`.EMB.PPC.sdata0`、`.EMB.PPC.sbss0` セクションだけです。

large データセクション用のデフォルトのアドレッシングモードは `far_abs` です。定義済み small データセクション用のデフォルトのアドレッシングモードは `sda_rel` です。

オプションのパラメータ `code_mode=codemode` は、セクションの実行可能ルーチンを参照するときのアドレッシングモードを指定します。

`codemode` で指定可能なアドレッシングモードを以下に示します。

`pc_rel` : 呼び出し元から 24 ビット以内にあるルーチン

`near_abs` : RAM の最初に 24 ビット以内にあるルーチン

実行可能コードセクション用のデフォルトのアドレッシングモードは `pc_rel` です。

---

注意： すべてのセクションはデータアドレッシングモード (`data_mode=datamode`) とコードアドレッシングモード (`code_mode=codemode`) を持ちます。エンベデッドシステム用 CodeWarrior C/C++ コンパイラは実行可能コードをデータセクションに保存し、データを実行可能コードセクションに保存することができますが、これはお奨めできません。

---

## セクションアクセス権

`#pragma section` でセクションを定義した場合、デフォルトのアクセス権はリードオンリーです。特定のオブジェクトタイプのためにカレントセクションを変更したい場合、コンパイラはアクセス権を調整してそのオブジェクトタイプの保存を許可します。また事前に許可されたオブジェクトタイプも保存します。`code_type` をセクションに付加すると、そのセクションへの実行許可が追加されます。`data_type`、`sdata_type`、`sconst_type` をセクションに付加すると、そのセクションへの書き込み許可が追加されます。

あるオブジェクトタイプ用にカレントセクションではないセクションを作成した場合、`__declspec` キーワードでオブジェクトをそのセクションへ入れることができます。コンパイラは自動的にそのセクションのアクセス権を更新し、オブジェクトの保存を許可します。次に警告を出します。この警告を出さないためには、オブジェクトコードを保存する前に、そのセクションに対して正式なアクセス権を与えてください。セクションにオブジェ

クトタイプが付加されている場合、既にそのアクセス権を持つセクションに再度アクセス権が与えられます。

注意： A セクションにオブジェクトタイプが付加されている場合、適切なアクセス権が設定されます。

### 定義済みセクションとデフォルトセクション

オブジェクトタイプで設定された定義済みセクションは、そのタイプのデフォルトセクションになります。標準以外のセクションをオブジェクトタイプに割り当てた後、[「#pragma section のフォーム」\(p138\)](#)のいずれかのフォームでデフォルトセクションを逆にすることができます。

コンパイラは事前にセクションを定義します ([例 6.7](#))。

#### 例 6.7 定義済みセクション

```
#pragma section code_type ".text" data_mode=far_abs \  
    code_mode=pc_rel  
#pragma section data_type ".data" ".bss" data_mode=far_abs \  
    code_mode=pc_rel  
#pragma section const_type ".rodata" ".rodata" data_mode=far_abs \  
    code_mode=pc_rel  
#pragma section sdata_type ".sdata" ".sbss" data_mode=sda_rel \  
    code_mode=pc_rel  
#pragma section sconst_type ".sdata2" ".sbss2" data_mode=sda_rel \  
    code_mode=pc_rel  
#pragma section ".EMB.PPC.sdata0" ".EMB.PPC.sbss0" \  
    data_mode=sda_rel code_mode=pc_rel
```

注意： ".EMB.PPC.sdata0" および ".EMB.PPC.sbss0" セクションは sdata\_type オブジェクトタイプの代わりとしても事前に定義できます。

### #pragma section のフォーム

このプラグマには以下のフォームがあります。

```
#pragma section ".name1"
```

これで ".name1" というセクションが作成できます (同名のものがない場合)。このフォームによりコンパイラは、#pragma section の後に適切なステートメントがない、または \_\_declspec キーワードで定義されたアイテムを持たないオブジェクトをこのセクションに保存しません。指定されているセクション名が 1 つだけなら、それは初期化済みオブジェクトセクション (iname) の名前とみなされます。セクションが既に定義されている場合、さらに未初期化オブジェクトセクション (uname) を指定できます。セクションに読み取

りと書き込みの許可が必要であれば、`".name1"` ではなく `#pragma section RW` を使います (特にキーワードを使っている場合)。

```
#pragma section objecttype ".name2"
```

複数のオブジェクトタイプを追加すると、コンパイラは `".name2"` セクションで指定されているタイプのオブジェクトを保存します。`".name2"` が存在しない場合、コンパイラは適切なアクセス権を設定してセクションを作成します。指定されているセクション名が1つだけなら、それは初期化済みオブジェクトセクション (`iname`) の名前とみなされます。セクションが既に定義されている場合、さらに未初期化オブジェクトセクション (`uname`) を指定できます。

```
#pragma section objecttype
```

パラメータ `iname` がない場合、コンパイラはオブジェクトタイプのセクションをデフォルトセクションへリセットします。定義済みセクションについては「[定義済みセクションとデフォルトセクション](#)」(p138) を参照してください。オブジェクトタイプのセクションのリセットによって、アドレッシングモードがリセットされることはありません。これは明示的に行わなくてはなりません。

セクションを宣言または設定するには、未初期化セクションを持たないセクションに追加します。未初期化セクションが既に付加されている場合、初期化セクションを持つ未初期化セクションを変更することはできません。初期化セクションに対応する未初期化セクションも同じです。

個々のオブジェクトを特定のセクションへ入れる

あるタイプの特定のオブジェクトを、カレントセクションを変更することなくそれ以外のセクションへ保存することができます。`__declspec` キーワードとターゲットセクションの名前を、`extern` 宣言か、セクションに保存したいアイテムの静的定義の後ろに追加します (例 6.8)。

#### 例 6.8 `__declspec` を使ってオブジェクトを特定のセクションへ入れる

```
__declspec(".data") extern int myVar;
#pragma section "constants"
__declspec("constants") const int myvar = 0x12345678;
```

`#pragma push`、`#pragma pop` と `#pragma section` を使う

`#pragma push`、`#pragma pop` と一緒にプラグマを使うことができます。例 6.6 を参照してください。`#pragma pop` は、既に存在している、または `#pragma push` に対応するセクションのアクセス権への変更を復帰しません。

## segment

説明 それ以降のオブジェクトコードをストアするセグメントを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma segment name`

注意 このプラグマは Mac OS プログラミングでのみ有効です。

このプラグマは、これに続くすべての関数を *name* という名のコードセグメントに置きます。関数単位のセグメンテーションについては、各『Targeting』マニュアルを参照してください。

PowerPC アプリケーションにはコードセグメントがないので、一般的に PowerPC コンパイラはこの疑似命令を無視します。しかし、PPC PEF 設定パネルの [ コード並び替え ] ポップアップメニューで [ #pragma segment を利用 ] を選択した場合、PowerPC コンパイラは、関数を同じセグメントに集めます。詳しくは、各『Targeting』マニュアルを参照してください。

このプラグマは設定パネルのどのオプションとも対応していません。

## side\_effects

説明 ポインタエイリアスの使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma side_effects on | off | reset`

注意 プログラムがポインタエイリアスを使用していないならば、プログラムをより小さく速くするためにこのプラグマを `off` にしてください。ポインタエイリアスを使っているなら、おかしなコードにならないようにこのプラグマを `on` にしてください。ポインタエイリアスは次のようなものです。

```
int a, *p;
p = &a;    // *p は a のエイリアス
```

なぜポインタエイリアスが重要かを理解するために、コンパイラは、計算を行う前に変数をレジスタに入れる必要があることを覚えておいてください。そして以下の例のように、コンパイラは、最初の加算の前に *a* をレジスタに入れます。*\*p* が *a* のエイリアスの場合、*\*p* を変えると *a* が変わってしまうので、2 つ目の加算の前に *a* を再びレジスタに入れなければいけません。*\*p* が *a* のエイリアスでなければ *\*p* を変えても *a* が変わらないので、コンパイラは *a* を再びレジスタに入れる必要はありません。

```
x = a + 1;
*p = 0;      // *p が a のエイリアスだと
y = a + 2;   //   これは a を変える
```

注意: PowerPC コンパイラはこのプラグマを無視し、常にプログラムがポインタエイリアスを含んでいると想定しています。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (side_effects)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトではこのプラグマは on です。

### simple\_prepdump

説明 プリプロセッサダンプのコメントの表示をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma simple_prepdump on | off | reset`

注意 デフォルトでは、プリプロセッサはプロセス中のカレントのインクルードファイルの出力に対してコメントを追加します。このプラグマを on にするとこれらのコメントを無効にすることができます。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option (simple_prepdump)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトではこのプラグマは on です。

### SOMCallOptimization

説明 SOM オブジェクトの呼び出しのエラーをチェックします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma SOMCallOptimization on | off | reset`

注意 このプラグマは C++ を使った Mac OS プログラミングでのみ有効です。

PowerPC コンパイラは、少し小さいけれども若干遅い最適化されたエラーチェックを使っています。

プラグマ `direct_to_SOM` が `off` の場合、このプラグマは無視されます (「[direct\\_to\\_som](#)」(p90))。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option` ([SOMCallOptimization](#)) を使います (「[オプションのチェック](#)」(p162))。デフォルトではこのプラグマは `off` です。

SOMCallStyle

説明 SOM オブジェクトの呼び出し規約を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma SOMCallStyle OIDL | IDL`

注意 プラグマ `SOMCallStyle` で、次のいずれかの SOM 呼び出し規約を選べます。

OIDL : DSOM をサポートしない古いスタイル

IDL : SOM をサポートする新しいスタイル

IDL スタイルを使ったクラスでは、そのメソッドは最初のパラメータに環境ポインタを持たなければいけません。SOMClass と SOMObject クラスは OIDL を使っていることに注意してください。それらのいずれかのメソッドをオーバーライドするときは、環境ポインタを含めることはできません。

このプラグマはプラグマ `direct_to_SOM` が `off` の場合、無視されます。『Targeting Mac OS』マニュアルを参照してください。

このプラグマに相当する設定パネルのオプションはありません。on か否かをチェックするには、`__option` ([SOMCheckEnvironment](#)) を使います (「[オプションのチェック](#)」(p162))。デフォルトではこのプラグマは IDL に設定されています。

SOMCheckEnvironment

説明 SOM 環境チェックを行うか否かを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     `#pragma SOMCheckEnvironment on | off | reset`

注意 このプラグマは C++ コードを使用する Mac OS にのみ有効です。

プラグマ `SOMCheckEnvironment` が `on` の場合、コンパイラは自動 SOM 環境チェックを行います。すべての IDL メソッドコールと `new` アロケーションを、エラーチェック関数も同時に呼ぶような表記にします。メソッド呼び出しと割り当てには異なるエラーチェック関数を定義しなければなりません。これらの関数の記述方法については『Targeting Mac OS』マニュアルを参照してください。

コンパイラは次の IDL メソッドコールを、

---

```
SOMobj->func(&env, arg1, arg2) ;
```

---

以下と等価であるようなものにします。

---

```
( temp=SOMobj->func(&env, arg1, arg2),  
  __som_check_ev(&env), temp ) ;
```

---

まずコンパイラはメソッドをコールし、その結果をテンポラリ変数にストアします。そして環境ポインタをチェックし、最後にメソッドの結果を戻します。

コンパイラは次の `new` アロケーションを、

---

```
new SOMclass;
```

---

以下と等価であるようなものにします。

---

```
( temp=new SOMclass, __som_check_new(temp), temp);
```

---

まずコンパイラはオブジェクトを作成し、それをテンポラリ変数にストアします。そしてオブジェクトをチェックし、それを戻します。

PowerPC コンパイラは、上記のものより少し小さいけれども若干遅い、最適化されたエラーチェックを使います。上に述べたエラーチェックを PowerPC コードで使うには、プラグマ `SOMCallOptimization` を使ってください。[「SOMCallOptimization」\(p141\)](#) を参照してください。

このプラグマはプラグマ `direct_to_SOM` が `off` の場合、無視されます。『Targeting Mac OS』マニュアルを参照してください。

このプラグマは C/C++ 言語設定パネルの [ Direct to SOM ] ポップアップメニューに相当します。このメニューの [ 環境チェックと共に利用 ] は、このプラグマを `on` にすることと等価です。それ以外のものは `off` にするのと等価です。`on` か否かをチェックするには、`__option (SOMCheckEnvironment)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトではこのオプションは `on` です。

## SOMClassVersion

説明 SOM クラスのバージョンを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma SOMClassVersion(class, majorVer, minorVer)`

注意 このプラグマは C++ を使った Mac OS プログラミングでのみ有効です。

SOM は、クラスが他のソフトウェアと互換性があるかどうかを判断するのに、クラスのバージョン番号を使います。バージョン番号を宣言しないと、SOM はゼロと判断します。バージョン番号はゼロか正の数でなければいけません。

クラスを定義するとき、プログラムはそのバージョン番号をクラスのメタデータ内の SOM カーネルに渡します。そのクラスのオブジェクトをインスタンス化するとき、プログラムはそのバージョン番号をランタイムカーネルに渡し、カーネルが、そのクラスが実行中のソフトウェアと互換性があるか否か確認します。

このプラグマは、プラグマ `direct_to_SOM` が `off` の場合は無視されます。『Targeting Mac OS』マニュアルを参照してください。

このプラグマに相当する設定パネルのオプションはありません。

## SOMMetaClass

説明 SOM クラスのメタクラスを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma SOMMetaClass (class, metaclass)`

注意 このプラグマは C++ を使った Mac OS プログラミングでのみ有効です。

メタクラスは特殊な SOM クラスで、他の SOM クラスのインプリメンテーションを定義するものです。メタクラス自身も含めて、すべての SOM クラスは、メタクラスを持っています。デフォルトでは、SOM クラスのメタクラスは `SOMClass` です。他のメタクラスを使いたい場合は、プラグマ `SOMMetaClass` を使ってください。

メタクラスは SOM クラスの子でなければいけません。また、クラスは自身のメタクラスでもいけません。すなわち、クラスとメタクラスは異なるクラス名でなければいけません。



このプラグマは、プラグマ `direct_to_SOM` が `off` の場合は無視されます。『Targeting Mac OS』マニュアルを参照してください。

このプラグマに相当する設定パネルのオプションはありません。

## SOMReleaseOrder

説明 SOM クラスのメンバー関数を解放する順番を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma SOMReleaseOrder(func1, func2, ... funcN)`

注意 このプラグマは C++ を使った Mac OS プログラミングでのみ有効です。

SOM クラスは、メンバー関数の解放順序を指定しなければいけません。最初にクラスを開発する時に便利のように、CodeWarrior C++ では、`SOMReleaseOrder pragma` を書かないと、解放順序はクラス宣言内の関数順とみなしてくれます。しかし、そのクラスのあるバージョンをリリースするときは、後のバージョンでリストを変更する必要があるので、このプラグマを使います。

クラスが作るすべての SOM メソッドを指定しなければいけません。仮想インラインメンバー関数を指定しないでください。なぜなら、それらは SOM メソッドとはみなされないからです。オーバーライドされる関数も指定しないでください。

後のバージョンのクラスで関数を削除するときは、その名前を解放順序リストに残しておいてください。関数を追加するときは、リストの最後に置いてください。関数のクラス階層を上げるときは、もとのリストはそのままにしておいて、新しいクラスのリストに追加してください。

このプラグマは、プラグマ `direct_to_SOM` が `off` の場合は無視されます。『Targeting Mac OS』マニュアルを参照してください。

このプラグマに相当する設定パネルのオプションはありません。

## stack\_cleanup

説明 スタックのクリーンアップを行うコードの生成をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma stack_cleanup on | off | reset`

**注意** `on` の場合、関数呼び出しの後の遅延型スタックのクリーンアップが無効になります。コンパイラに関数呼び出しの後にスタックから引数を削除させます。このオプションは実行速度を下げるため、スタックの使用を減らし、スタックがプログラムの他のパーツへ侵入する可能性が少なくなります。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (stack_cleanup)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトは `off` です。

### `static_inlines`

**説明** コンパイラが生成するインライン関数のインスタンスの数をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma static_inlines on | off | reset`

**注意** プラグマ `static_inlines` は、インライン宣言された関数をインライン呼び出しできないけれども関数をコンパイルしたバージョンを作らなければいけないときに、コンパイラがどう対処するかを決定します。このプラグマが `off` の場合、コンパイラはすべてのプロジェクトをコンパイルした 1 つのバージョンを作ります。`on` の場合は、コンパイラは、それぞれのファイルに対して異なるコンパイルされたバージョン（これはコンパイル済みバージョンを必要とする）を作ります。

このプラグマは、コンパイラがある有効なスイートを渡せるときだけ有効です。一般に、コードをスピードの低下なしに小さくするために、このプラグマを `off` のままにしておきます。

このプラグマに相当する設定パネルのオプションはありません。`on` か否かをチェックするには、`__option (static_inlines)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトは `off` です。

### `suppress_init_code`

**説明** 静的初期化オブジェクトコードの生成をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma suppress_init_code on | off | reset`

**注意** このプラグマが `on` の場合、コンパイラは静的データ初期化コード (C++ コンストラクタ関数など) を生成しません。デフォルトは `off` です。

**警告!** このプラグマはプログラムに予期せぬ動作をさせることがあるので、注意が必要です。

`sym`

**説明** デバッグシンボル情報の生成をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma sym on | off | reset`

**注意** コンパイラは、IDE のプロジェクトウィンドウでデバッグ列のマークをオンにしたときだけこのプラグマに注意を払います。このプラグマが `off` の場合、コンパイラはソースファイルのデバッグシンボルファイル (SYM または DWARF) に対して、プラグマ以降の関数のデバッグ情報を出力しません。`on` の場合、コンパイラはデバッグ情報を生成します。

プロジェクトウィンドウ内のソースファイルの隣にデバッグマークがあれば、コンパイラは、そのソースファイルに対する SYM ファイルを常に作成することに注意してください。このプラグマは、その SYM ファイル内にどの関数の情報を生成するのかを指定します。

このプラグマが `on` か否かをチェックするには、`__option (sym)` を使います([「オプションのチェック」\(p162\)](#))。デフォルトは `on` です。

`syspath_once`

**説明** インクルードファイルの処理方法を指定します。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma syspath_once on | off | reset`

**注意** このプラグマが `on` の場合、`#include <>` および `#include ""` 疑似命令で参照するファイルが同じであっても、別のファイルとして扱います。

## toc\_data

説明 静的変数の保存方法を指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma toc_data on | off | reset`

注意 このプラグマは Mac OS CFM プログラミングでのみ有効です。

プラグマ `toc_data` が `on` の場合、コンパイラはコードを小さくかつ速くします。どこかにスペースを確保して静的変数を保存し、そのポインタを TOC に置く代わりに、4 バイトまたはそれより小さな静的変数を TOC 内に直接入れます。このプラグマは、コードが TOC 内にデータへのポインタがあることを想定している時にだけ `off` にしてください。

このプラグマは PPC プロセッサ設定パネルの [ 小さな Static データを TOC 内に保存する ] オプションに相当します。on か否かをチェックするには、`__option (toc_data)` を使います (「[オプションのチェック](#)」(p162))。

## trigraphs

説明 ANSI/ISO の Trigraph 拡張の使用をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma trigraphs on | off | reset`

注意 ANSI 規格に厳密に従ったコードを書くときは、C/C++ 言語設定パネルで、このプラグマ `trigraphs` を `on` にしてください。多くの Mac OS で一般的な文字は `trigraph` シーケンスのように見えるので、このプラグマを使えば、それらをエスケープキャラクタなしで使うことができます。文字列や複数文字の定数を、`?` マークを含むもので初期化する時に注意が必要です。

---

```
char c = '????'; // ERROR: Trigraph シーケンス '??' に展開される
char d = '\? \? \? \?'; // OK
```

---

このプラグマは [C/C++ 言語設定パネル](#) の [ Trigraph 拡張 ] オプションに相当します。on か否かをチェックするには、`__option (trigraphs)` を使います (「[オプションのチェック](#)」(p162))。

## traceback

説明 AIX フォーマットのデバッグ用トレースバックテーブルを生成します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma traceback on | off | reset`

注意 ソースコードを公開しないアプリケーションや共有ライブラリを第三者がデバッグするとき、このプラグマが役立ちます。このプラグマが `on` の場合、コンパイラは各関数に対して AIX フォーマットのトレースバックテーブルを作成し、これを実行コード内に置きます。CodeWarrior デバッガも Apple 社のデバッガも、このトレースバックテーブルを扱うことができます。

このプラグマは PPC プロセッサ設定パネルの [トレースバックテーブル] ポップアップメニューに相当します。on か否かをチェックするには、`__option (traceback)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトは `off` です。

## unsigned\_char

説明 `char` 型宣言を `unsigned char` として扱います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma unsigned_char on | off | reset`

注意 プラグマ `unsigned_char` が `on` の場合、C/C++ コンパイラは `char` 宣言を `unsigned char` 宣言と同じものとして扱います。

---

注意： このプラグマが `on` の場合、このプラグマが `off` でコンパイルされたライブラリと互換性がなくなります。そのコードは CodeWarrior の ANSI ライブラリと共に動かないかもしれません。

---

このプラグマは C/C++ 言語設定パネルの [unsigned char を利用] オプションに相当します。on か否かをチェックするには、`__option (unsigned_char)` を使います ([「オプションのチェック」\(p162\)](#))。デフォルトではこのオプションは `off` です。

## unused

説明 関数内の参照されていない変数や引数への警告の表示をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma unused ( var_name [, var_name ]... )`

**注意** このプラグマは、引数リスト内の未使用値や未使用パラメータに対するコンパイル時のエラーを抑制します。このプラグマは関数の内部でだけ使用可能で、列挙される変数はその関数のスコープ内になければいけません。このプラグマは、クラス定義内で定義された関数や、テンプレート関数に対しては使えません。

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)
{
    int b;
#pragma unused(a,b)  // コンパイラは a と b が使われて
                      // いないことに文句を言わない

    // . . .
}
```

このプラグマに相当する設定パネルのオプションはありません。

### use\_fp\_instructions

**説明** NEC V800 浮動小数点関数の生成をコントロールします。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ** `#pragma use_fp_instructions on|off|reset`

**注意** このプラグマは NEC V800 Processor 設定パネルの [ Use V810 Floating-Point Instructions ] オプションに相当します。on か否かをチェックするには、`__option (use_fp_instructions)` を使います ([「オプションのチェック」\(p162\)](#))。

### use\_frame

**説明** スタックフレームに BP レジスタを使います。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma use_frame on|off|reset`

注意 このプラグマが on の場合、コンパイラは BP レジスタを使ってスタックフレームの開始点を指します。

このプラグマが on か否かをチェックするには、`__option (use_frame)` を使います([「オプションのチェック」\(p162\)](#))。

`use_mask_registers`

説明 NEC V800 r20 および r21 レジスタを使います。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma use_mask_registers on|off|reset`

注意 このプラグマは NEC V800 Processor 設定パネルの [ Use r20 and r21 as Mask Registers ] オプションに相当します。on か否かをチェックするには、`__option (use_mask_registers)` を使います ([「オプションのチェック」\(p162\)](#))。

`warn_emptydecl`

説明 変数を伴わない宣言の認識をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_emptydecl on | off | reset`

注意 プラグマ `warn_emptydecl` が on の場合、コンパイラは変数のない宣言を見つけると警告を出します。

---

```
int ;      // WARNING
int i;     // OK
```

---

このプラグマは C/C++ 警告設定パネルの [ 識別子がない宣言 ] オプションに相当します。on か否かをチェックするには、`__option (warn_emptydecl)` を使います([「オプションのチェック」\(p162\)](#))。

## warning\_errors

説明 警告をエラーとして扱うか否かを指定します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warning_errors on | off | reset`

注意 プラグマ `warning_errors` が `on` の場合、コンパイラはすべての警告をエラーとして扱います。すべての警告が解決されるまで、コンパイラはファイルをコンパイルしません。

このプラグマは C/C++ 警告設定パネルの [ 警告をエラーとして扱う ] オプションに相当します。`on` か否かをチェックするには、`__option (warning_errors)` を使います ([「オプションのチェック」\(p162\)](#))。

## warn\_extracomma

説明 不要なコンマに対して警告を出します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_extracomma on | off | reset`

注意 プラグマ `warn_extracomma` が `on` の場合、コンパイラは不要なコンマを見つけると警告を出します。次の例は C では正しいですが、`on` の場合は警告が出ます。

```
int a[] = { 1, 2, 3, 4, }; // ^ WARNING:4 の後に不要なコンマ
```

このプラグマは C/C++ 警告設定パネルの [ 警告をエラーとして扱う ] オプションに相当します。`on` か否かをチェックするには、`__option (warn_extracomma)` を使います ([「オプションのチェック」\(p162\)](#))。

## warn\_hidevirtual

説明 スーパークラスの仮想関数を隠している非仮想メンバー関数の扱いをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------



プロトタイプ `#pragma warn_hidevirtual on|off|reset`

**注意** プラグマ `warn_hidevirtual` が `on` の場合、スーパークラスの仮想関数を隠す非仮想メンバー関数を宣言する場合、コンパイラは警告を出します。同じ名前だが引数の型の異なる関数は、他の関数を隠します。

---

```
class A {
public:
    virtual void f(int);
    virtual void g(int);
};

class B: public A {
public:
    void f(char);           // WARNING:A::f(int) を隠す
    virtual void g(int);    // OK:A::g(int) をオーバーライドする
};
```

---

このプラグマは C/C++ 警告設定パネルの [ 仮想関数が隠ぺいされた場合 ] オプションに相当します。on か否かをチェックするには、`__option (warn_hidevirtual)` を使います (「[オプションのチェック](#)」(p162))。デフォルトは `off` です。

## warn\_illpragma

**説明** 不当なプラグマ疑似命令に警告を出します。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_illpragma on | off | reset`

**注意** プラグマ `warn_illpragma` が `on` の場合、コンパイラは不当なプラグマを見つけると警告を出します。

---

```
#pragma near_data off    // WARNING:near_data は pragma ではない
#pragma far_data select  // WARNING:select は定義されていない
#pragma far_data on      // OK
```

---

このプラグマは C/C++ 警告設定パネルの [ 不当な Pragma ] オプションに相当します。on か否かをチェックするには、`__option (warn_illpragma)` を使います (「[オプションのチェック](#)」(p162))。

## warn\_implicitconv

説明 暗黙の算術変換に対して警告を出します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_implicitconv on | off | reset`

注意 ソースの値がデスティネーションの型で表せないときに、暗黙の算術変換に対してコンパイラは警告を出します ( [例 6.9](#) )。

### 例 6.9 暗黙の算術変換の例

```
#pragma warn_implicitconv on

char foo(int a)
{
    return a+1; // Warning : 'int' から 'char' への暗黙の算術変換
}
```

このプラグマは C/C++ 警告設定パネルの [ 暗黙の算術変換 ] オプションに相当します。on か否かをチェックするには、`__option (warn_implicitconv)` を使います ( [「オプションのチェック」\( p162 \)](#) )。

## warn\_no\_side\_effect

説明 冗長ステートメントへの警告の表示をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_no_side_effect on | off | reset`

注意 on の場合、コンパイラは副作用を生じないステートメントを発見すると警告を発します。副作用のないステートメントに対する警告を抑止するには、ステートメントを (void) でキャストします。 [例 6.10](#) を参照してください。

### 例 6.10 pragma warn\_no\_side\_effect の使用例

```
#pragma warn_no_side_effect on
void foo(int a,int b)
{
```

```

    a+b; // warning: expression has no side effect
    (void)(a+b); // void cast suppresses warning
}

```

on か否かをチェックするには `__option (warn_no_side_effect)` を使います([「オプションのチェック」\(p162\)](#))。

## warn\_notinlined

説明 関数をインラインできないときに警告を出します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_notinlined on | off | reset`

注意 関数をインラインできないときに、コンパイラは警告を出します。このプラグマは C/C++ 警告設定パネルの「関数がインライン展開されなかった場合」オプションに相当します。on か否かをチェックするには `__option (warn_notinlined)` を使います([「オプションのチェック」\(p162\)](#))。

## warn\_padding

説明 構造体へのパディングに関する通知をコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ `#pragma warn_padding on | off | reset`

注意 `warn_padding` が on の場合、メモリ内のアライメントを向上させるために構造体へ追加したバイト数についてコンパイラは警告を出します。データ構造体へのパッド方法については各『Targeting』マニュアルを参照してください。

このプラグマが on か否かをチェックするには `__option (warn_padding)` を使います([「オプションのチェック」\(p162\)](#))。デフォルトは off です。

## warn\_possunwant

説明 意図的ではない論理的エラーに対して警告を出します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma warn\_possunwant on | off | reset

**注意** プラグマ `warn_possunwant` が `on` の場合、コンパイラは、C/C++ では正しいけれども予期せぬ副次効果を生むかもしれない、例えば意図的ではないセミコロンの挿入や `=` と `==` の勘違いなどの、よくある入力ミスをチェックします。コンパイラは、次のいずれかを見つけると警告を出します。

`if`、`while`、`for` での論理表現や条件中の代入。このチェックは `==` を使うところで `=` をしばしば使ってしまう人には有効です。

```
if (a=b) f();           // WARNING: a=b は代入
if ((a=b)!=0) f();      // OK: (a=b)!=0 は比較
if (a==b) f();          // OK: (a==b) は比較
```

文中の等価比較。このチェックは `=` を使うところで `==` を使ってしまう人には有効です。

```
a == 0;      // WARNING: これは比較
a = 0;       // OK: これは代入
```

`while`、`if`、`for` 直後のセミicolon (;)。次の例はエラーになり、おそらく意図的ではない無限ループになります。

```
while (i++); // WARNING: 意図的でない無限ループ
```

意図的に無限ループを作りたいときは、`while` 文とセミcolonのあいだにスペースかコメントを入れてください。

```
while (i++) ;      // OK: スペース挿入
while (i++) /* OK: コメント挿入 */ ;
```

このプラグマは C/C++ 警告設定パネルの [ 間違え易いエラー ] オプションに相当します。`on` か否かをチェックするには、`__option (warn_possunwant)` を使います([「オプションのチェック」\(p162\)](#))。

### warn\_resultnotused

**説明** 関数の結果が無視されたときに警告を表示します。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma warn\_resultnotused on | off | reset

注意 on の場合、コンパイラは関数を呼び出しているが、その結果を使わないステートメントを発見したときに警告を出します。警告を抑止するには、ステートメントを (void) でキャストします。例 6.11 を参照してください。

例 6.11     #pragma warn\_resultnotused の使用例

```
#pragma warn_resultnotused on
void foo(int a,int b)
{
    bar(); // warning: bar()'s result is not used
    (void)bar(); // void cast suppresses warning
}
```

on か否かをチェックするには、\_\_option (warn\_resultnotused) を使います(「[オプションのチェック](#)」(p162))。

warn\_structclass

説明 class と struct キーワードの意図的ではない混合に対して警告を発します。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma warn\_structclass on | off | reset

注意 プラグマ warn\_structclass が on の場合、コンパイラは class と struct キーワードが同じ識別子の宣言と定義で使われているときに警告を出します。

```
class X;
struct X { int a; }; // warning
```

このプラグマは C/C++ 警告設定パネルの[ キーワード 'class' と 'struct' の一貫性のない使用 ] オプションに相当します。on か否かをチェックするには、\_\_option (warn\_structclass) を使います (「[オプションのチェック](#)」(p162))。

warn\_unusedarg

説明 参照されない引数に対して警告を出します。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ**     #pragma warn\_unusedarg on | off | reset

**注意** プラグマ `warn_unusedarg` が `on` の場合、宣言されているが使われていない引数を見つければ、コンパイラが警告を出します。このチェックは、引数名のミススペルや引数名忘れなどを見つけるのに役立ちます。

---

```
void foo(int temp, int error)
{
    error = do_something(); // ERROR: error は定義されていない
} // WARNING: temp と error は使われていない
```

---

このプラグマは C/C++ 警告設定パネルの [ 未使用引数 ] オプションに相当します。on か否かをチェックするには、\_\_option (warn\_unusedarg) を使います (「[オプションのチェック](#)」(p162))。

### warn\_unusedvar

**説明** 参照されない変数に対して警告を出します。

**互換性** このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

**プロトタイプ**     #pragma warn\_unusedvar on | off | reset

**注意** プラグマ `warn_unusedvar` が `on` の場合、宣言されているけれども使われていない変数を見つければ、コンパイラは警告を出します。このチェックは変数名のミススペルや変数名忘れなどを見つけるのに役立ちます。

---

```
void foo(void)
{
    int temp, error;
    error = do_something(); // ERROR: error は定義されていない
} // WARNING: temp と error は使われていない
```

---

このプラグマは C/C++ 警告設定パネルの [ 未使用変数 ] オプションに相当します。on か否かをチェックするには、\_\_option (warn\_unusedvar) を使います (「[オプションのチェック](#)」(p162))。

warning

説明 互換性確保のためにのみ含まれています。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma warning(warning\_specifier : warning\_number\_list)

注意 このプラグマは x86 プログラミングでのみ有効です。

無視されます。Microsoft との互換性確保のために含まれています。  
warning\_number\_list はスペースで区切られた警告の数です。以下は  
warning\_specifier の例です。

once  
default  
1  
2  
3  
4  
disable  
error

wchar\_type

説明 wchar\_t 型のサイズとフォーマットをコントロールします。

互換性 このプラグマは以下のプラットフォームターゲットで有効です。

68K	PowerPC	NEC V800	Intel x86	MIPS
-----	---------	----------	-----------	------

プロトタイプ     #pragma wchar\_type on | off | reset

注意 プラグマ wchar\_type が on の場合、wchar\_t はビルトインタイプ ( 符号なし 16 ビット  
整数型として実装されている ) として扱われます。プラグマが off の場合、wchar\_t と文  
字列リテラルのキャラクタは unsigned short として扱われます。

このプラグマは C/C++ 言語設定パネルの [ wchar\_t 型サポート ] オプションに相当します。  
on か否かをチェックするには、\_\_option (wchar\_type) を使います (「[オプションの  
チェック](#)」(p162))。

## 定義済みシンボル

Metrowerks C/C++ は、コンパイル時の環境を伝えるために、いくつかのプリプロセッサシンボルを定義しています。これらのシンボルは、実行時ではなくコンパイル時に評価されることに注意してください。この節では以下の内容について説明します。

[ANSI の定義済みシンボル](#)

[Metrowerks の定義済みシンボル](#)

### ANSI の定義済みシンボル

以下の表に、ANSI C 規格に必要とされるシンボルをまとめます。

表 6.7 ANSI 定義済みシンボル

マクロ	意味
<code>__DATE__</code>	ファイルがコンパイルされた日付。例: "Jul 14, 1995".
<code>__FILE__</code>	コンパイルされたファイルの名前。例: "prog.c"
<code>__LINE__</code>	コンパイルされた行番号。ヘッダファイルをインクルードする前の番号。
<code>__TIME__</code>	ファイルがコンパイルされた時間の 24 時間表記。例: "13:01:45"
<code>__STDC__</code>	C ソースコードをコンパイルしたときには 1。このマクロは、Metrowerks C が ANSI C 規格を実装していることを知らせる。C++ ソースコードをコンパイルしたときはこのマクロは定義されない。

[例 6.12](#) に ANSI 定義済みシンボルを使った小さなプログラムを示します。

例 6.12 ANSI の定義済みシンボルの使用例

```
#include <stdio.h>

void main(void)
{
    printf("Hello World!\n");

    printf("%s, %s\n", __DATE__, __TIME__);
    printf("%s, line: %d\n", __FILE__, __LINE__);
}
```

このプログラムは次のように印字します。

```
Hello World!
Oct 31 1995, 18:23:50
main.ANSI.c, line: 10
```



## Metrowerks の定義済みシンボル

次の表に、Metrowerks C/C++ の追加シンボルを示します。

表 6.8 Metrowerks の定義済みシンボル

マクロ	意味
<code>__A5__</code>	データが A5 相対のときは 1、A4 相対のときは 0。68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__cplusplus</code>	このファイルを C++ ファイルとしてコンパイルする場合は定義され、C ファイルとしてコンパイルする場合は定義されない。
<code>__embedded_cplusplus</code>	Embedded C++ モードがオンの場合 1 として定義される。Embedded C++ がオフの場合、定義されない。
<code>__fourbyteints__</code>	68K プロセッサ設定パネルの [ 4 バイト Int ] オプションがオンの場合 1、オフの場合 0。これは 68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__IEEEdoubles__</code>	68K プロセッサ設定パネルの [ 8 バイト Doubles ] オプションがオンの場合 1、オフの場合 0。これは 68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__INTEL__</code>	コードを x86 コンパイラでコンパイルしているとき 1、そうでなければ 0。他のプラットフォームでは定義されていない。
<code>__MC68K__</code>	コードを 68K コンパイラでコンパイルしているとき 1、そうでなければ 0。これは 68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__MC68020__</code>	68K プロセッサ設定パネルの [ 68020 コード生成 ] オプションがオンの場合 1、オフの場合 0。PowerPC コンパイラでは定義されていない。これは 68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__MC68881__</code>	68K プロセッサ設定パネルの [ 浮動小数点モデル ] ポップアップメニューの [ 68881 ] オプションがオンの場合 1、オフの場合 0。これは 68K コンパイラ専用で、他のプラットフォームでは定義されていない。
<code>__MIPS__</code>	MIPS コンパイラでは 1、他のプラットフォームでは定義されていない。
<code>__MIPS_ISA2__</code>	コンパイラのターゲットが MIPS で、MIPS Processor 設定パネルの [ ISA II ] がオンの場合は 1。オフの場合、また他のターゲットでは定義されていない。

マクロ	意味
__MIPS_ISA3__	コンパイラのターゲットが MIPS で、MIPS Processor 設定パネルの [ ISA III ] がオンの場合は 1。オフの場合、また他のターゲットでは定義されていない。
__MIPS_ISA4__	コンパイラのターゲットが MIPS で、MIPS Processor 設定パネルの [ ISA IV ] がオンの場合は 1。オフの場合、また他のターゲットでは定義されていない。
__MWBROWSER__	CodeWarrior ブラウザがコードを解析していれば 1、そうでなければ 0。
__MWERKS__	CodeWarrior C/C++ コンパイラ (CW7 以降) のバージョン番号。例えば Metrowerks C/C++ のバージョンが 2.2 ならば、__MWERKS__ は 0x2200。CW7 以前のバージョンでは 1。
__profile__	68K プロセッサ設定パネルの [ プロファイラ情報を生成 ] オプションがオンの場合 1、オフの場合 0。
__powerc, powerc, __POWERPC__	コードを PowerPC コンパイラでコンパイルしている場合 1、そうでなければ 0。
macintosh	コードを 68K または Power Macintosh コンパイラでコンパイルしている場合 1、そうでなければ 0。

## オプションのチェック

プリプロセッサ関数 `__option()` を使うと、C/C++ コンパイラやコード生成に関係のあるプラグマやオプションの設定をチェックできます。これらの設定は一般的にはプロジェクト設定ダイアログで行います。

文法：

```
__option(option-name)
```

指定されたオプションが on の場合は `__option ()` は 1 を返し、それ以外の場合は 0 を返します。`option-name` が認識されない場合、`__option ()` は偽を返します。

この関数は 1 つのソースファイルの中に、異なるオプション設定向けのコードを入れたいときに役立ちます。次の例では、MC68881 浮動小数点ユニットを持つマシンと持たないマシン向けにどうコンパイルするかを示しています。

```
#if __option (code68881) // FPU 向けに最適化されたコード
#else // すべての 68K プロセッサ向けのコード
#endif
```

次の表にプリプロセッサ関数 `__option()` で使用できるすべてのオプション名をまとめ

ます。

オプション	相当する項目
a6frames(68K only)	68K プロセッサ設定パネルの [ A6 スタックフレームを生成 ] オプション、pragma a6frames
align_array_members	pragma align_array_members
always_inline	pragma always_inline.
altivec_codegen	Pragma altivec_codegen
altivec_model	Pragma altivec_model
altivec_vrsave	Pragma altivec_vrsave
ANSI_strict	<a href="#">C/C++ 言語設定パネル</a> の [ ANSI に厳格に従う ] オプション、PRAGMA ANSI_strict
arg_dep_lookup	pragma arg_dep_lookup
ARM_conform	<a href="#">C/C++ 言語設定パネル</a> の [ ARM に適合 ] オプション、pragma ARM_conform
auto_inline	<a href="#">C/C++ 言語設定パネル</a> の [ 自動インライン展開 ] オプション、pragma auto_inline
bool	<a href="#">C/C++ 言語設定パネル</a> の [ bool 型サポート ] オプション、pragma bool
check_header_flags	pragma check_header_flags
code68020	68K プロセッサ設定パネルの [ 68020 コード生成 ] オプション、pragma code68020
code68881	68K プロセッサ設定パネルの [ 浮動小数点モデル ] ポップアップメニューの [ 68881 ] オプション、pragma code68881
cplusplus	C++ ファイルとしてファイルをコンパイルする。 <a href="#">C/C++ 言語設定パネル</a> の [ C++ コンパイラを必ず利用 ] オプション、pragma cplusplus、およびマクロ cplusplus
cpp_extensions	praguma cpp_extensions
d0_pointers	praguma pointers_in_D0 と pointers_in_A0
def_inherited	praguma def_inherited.
defer_codegen	praguma defer_codegen.
direct_destruction	<a href="#">C/C++ 言語設定パネル</a> の [ C++ 例外処理有効 ] オプション、praguma direct_destruction
direct_to_SOM	<a href="#">C/C++ 言語設定パネル</a> の [ Direct to SOM ] ポップアップメニュー、praguma direct_to_SOM

オプション	相当する項目
disable_registers	praguma disable_registers
dollar_identifiers	praguma dollar_identifiers.
dont_inline	<a href="#">C/C++ 言語設定パネル</a> の [ インラインの深さ ] ポップアップメニューの [ インライン化しない ] praguma dont_inline
dont_reuse_strings	<a href="#">C/C++ 言語設定パネル</a> の [ 文字列定数を再利用しない ] オプション、praguma dont_reuse_strings
ecplusplus	praguma ecplusplus
EIPC_EIPSW	praguma EIPC_EIPSW
enumsalwaysint	<a href="#">C/C++ 言語設定パネル</a> の [ Enum は常に Int 型 ] オプション、praguma enumsalwaysint
exceptions	<a href="#">C/C++ 言語設定パネル</a> の [ C++ 例外処理有効 ] オプション、praguma exceptions
export	praguma export
extended_errorcheck	C/C++ 警告設定パネルの [ 拡張エラーチェック ] オプション、praguma extended_errorcheck
far_data	68K プロセッサ設定パネルの [ Far データ ] オプション、praguma far_data
far_strings	68K プロセッサ設定パネルの [ Far 文字列定数 ] オプション、praguma far_strings
far_vtables	68K プロセッサ設定パネルの [ Far メソッドテーブル ] オプション、praguma far_vtables
faster_pch_gen	praguma faster_pch_gen
float_constants	praguma float_constants
force_active	praguma force_active
fourbyteints	68K プロセッサ設定パネルの [ 4 バイト Int ] オプション、praguma fourbyteints
fp_contract	PPC プロセッサ設定パネルの [ FMADD と FMSUB を使用 ] オプション、praguma fp_contract
fullpath_prepdump	Pragma fullpath_prepdump.
gcc_extensions	Pragma gcc_extensions.
global_optimizer	グローバル設定パネルのオプション、praguma global_optimizer
IEEEdoubles	68K プロセッサ設定パネルの [ 8 バイト Doubles ] オプション、pragma IEEEdoubles

オプション	相当する項目
ignore_oldstyle	pragma ignore_oldstyle
import	pragma import.
inline_intrinsics	pragma inline_intrinsics
internal	pragma internal
interrupt	pragma interrupt
k63d	x86 プロセッサ設定パネルの [ 拡張命令 ] 欄の [ 3D Now! ] オプション、pragma k63d
k63d_calls	x86 プロセッサ設定パネルの [ 拡張命令 ] 欄の [ MMX ] と [ 3D Now! ] オプション、pragma k63d_calls
lib_export	pragma lib_export
line_prepdump	Pragma line_prepdump
little_endian	相当するオプションはなし。リトルエンディアンターゲット (x86 など) の場合 1。ビッグエンディアンターゲット (Mac OS など) の場合 0。
longlong	pragma longlong.
longlong_enums	pragma longlong_enums.
longlong_prepval	pragma longlong_enums.
no_static_dtors	pragma no_static_dtors.
macsbug	68K プロセッサ設定パネルの [ MacsBug シンボル ] ポップアップメニュー、pragma macsbug
microsoft_exceptions	pragma microsoft_exceptions
microsoft_RTTI	pragma microsoft_RTTI
mmx	x86 プロセッサ設定パネルの [ 拡張命令 ] 欄の [ MMX ] praguma mmx
mmx_call	pragma mmx_call
mpwc	68K プロセッサ設定パネルの [ MPW C 呼び出し規約 ] オプション、pragma mpwc
mpwc_newline	<a href="#">C/C++ 言語設定パネル</a> の [ CR の代わりに NL を利用 ] オプション、pragma mpwc_newline
mpwc_relax	<a href="#">C/C++ 言語設定パネル</a> の [ ポインタタイプルールを緩める ] オプション、pragma mpwc_relax
no_register_coloring	グローバル最適化設定パネルの [ 大域レジスタ割り当て ] オプション、pragma no_register_coloring
oldstyle_symbols	68K プロセッサ設定パネルの [ MacsBug シンボル ] ポップアップメニュー、pragma oldstyle_symbols

オプション	相当する項目
only_std_keywords	<a href="#">C/C++ 言語設定パネル</a> の [ ANSI キーワードのみ ] オプション、pragma only_std_keywords
opt_common_subs	pragma opt_common_subs
opt_dead_assignments	pragma opt_dead_assignments
opt_dead_code	pragma opt_dead_code
opt_lifetimes	pragma opt_lifetimes
opt_loop_invariants	pragma opt_loop_invariants
opt_propagation	pragma opt_propagation
opt_strength_reduction	pragma opt_strength_reduction
opt_unroll_loops	pragma opt_unroll_loops
opt_vectorize_loops	pragma opt_vectorize_loops
pool_data	PPC Processor 設定パネルの [ Pool Data ] オプション ( エンベデッド PowerPC プログラミングのみ )、pragma pool_data
pool_strings	<a href="#">C/C++ 言語設定パネル</a> の [ 文字列定数を一ヶ所にまとめる ] オプション、pragma pool_strings
ppc_unroll_speculative	Pragma ppc_unroll_speculative
precompile	ファイルがプリコンパイルされているか否か
preprocess	ファイルがプリプロセスされているか否か
profile	68K プロセッサ設定パネルの [ プロファイラ情報を生成 ] オプション、PPC プロセッサ設定パネルの [ プロファイラ情報を生成 ] オプション、pragma profile
readonly_strings	PPC プロセッサ設定パネルの [ 文字列を読み取り専用にする ] オプション、pragma readonly_strings
register_coloring	praguma register_coloring
require_prototypes	<a href="#">C/C++ 言語設定パネル</a> の [ 関数プロトタイプが必要 ] オプション、pragma require_prototypes
RTTI	<a href="#">C/C++ 言語設定パネル</a> の [ Enable RTTI 有効 ] オプション、pragma RTTI
side_effects	pragma side_effects
simple_prepdump	pragma simple_prepdump
SOMCallOptimization	pragma SOMCallOptimization
SOMCheckEnvironment	<a href="#">C/C++ 言語設定パネル</a> の [ Direct to SOM ] ポップアップメニュー、pragma SOMCheckEnvironment

オプション	相当する項目
static_inlines	pragma static_inlines
stack_cleanup	pragma stack_cleanup
suppress_init_code	pragma suppress_init_code
sym	プロジェクトウィンドウのデバッグ列のマーカ、 pragma sym
syspath_once	pragma syspath_once.
toc_data	PPC プロセッサ設定パネルの [ 小さな Static データを TOC 内に保存する ] オプション、pragma toc_data
traceback	pragma traceback
trigraphs	<a href="#">C/C++ 言語設定パネル</a> の [ Trigraph 拡張 ] オプション、 pragma trigraphs
unsigned_char	<a href="#">C/C++ 言語設定パネル</a> の [ unsigned char を利用 ] オプ ション、pragma unsigned_char
use_fp_instructions	NEC V800 Processor 設定パネルの [ Use V810 Floating- Point Instructions ] オプション、pragma use_fp_instructions
use_frame	pragma use_frame.
use_mask_registers	NEC V800 Processor 設定パネルの[ Use r20 and r21 as Mask Registers ] オプション、 praguma use_mask_registers
warn_emptydecl	C/C++ 警告設定パネルの [ 識別子がない宣言 ] オプシ ョン、pragma warn_emptydecl
warn_extracomma	C/C++ 警告設定パネルの [ 余分なコンマ ] オプション、 pragma warn_extracomma
warn_hidevirtual	C/C++ 警告設定パネルの [ 仮想関数が隠ぺいされた場合 ] オプション、pragma warn_hidevirtual
warn_illpragma	C/C++ 警告設定パネルの [ 不当な Pragma ] オプション、 pragma warn_illpragma
warn_implicitconv	C/C++ 警告設定パネルの [ 暗黙の算術変換 ] オプション、 pragma warn_implicitconv
warn_no_side_effect	pragma warn_no_side_effect
warn_notinlined	C/C++ 警告設定パネルの [ 関数がインライン展開されな かった場合 ] オプション、pragma warn_notinlined
warn_padding	pragma warn_padding
warn_possunwant	C/C++ 警告設定パネルの [ 間違え易いエラー ] オプシ ョン、pragma warn_possunwant

オプション	相当する項目
warn_resultnotused	pragma warn_resultnotused
warn_structclass	C/C++ 警告設定パネルの [ キーワード 'class' と 'struct' の一貫性のない使用 ] オプション、 pragma warn_structclass
warn_unusedarg	C/C++ 警告設定パネルの [ 未使用引数 ] オプション、 pragma warn_unusedarg
warn_unusedvar	C/C++ 警告設定パネルの [ 未使用変数 ] オプション、 pragma warn_unusedvar
warning_errors	C/C++ 警告設定パネルの [ 警告をエラーとして扱う ] オプション、 pragma warning_errors
wchar_type	C/C++ 言語設定パネルの [ wchar_t 型サポート ] オプション、 pragma wchar_type



# 索引

---

## 記号

#else 35  
#endif 35  
#include directive 102  
#line directive 111  
#pragma ステートメント 72  
# とマクロ 35  
=  
    意図しない 20  
=  
    オペレータ 50  
\_\_A5\_\_ 161  
\_\_builtin\_align() 44  
\_\_builtin\_type() 44  
\_\_cplusplus 161  
\_\_DATE\_\_ 160  
\_\_embedded\_cplusplus 65, 161  
\_\_FILE\_\_ 160  
\_\_fourbyteints\_\_ 161  
\_\_IEEEdouble\_\_ 161  
\_\_INTEL\_\_ 161  
\_\_LINE\_\_ 160  
\_\_MC68020\_\_ 161  
\_\_MC68881\_\_ 161  
\_\_MC68K\_\_ 161  
\_\_MIPS\_\_ 161  
\_\_MIPS\_ISA2\_\_ 161  
\_\_MIPS\_ISA3\_\_ 162  
\_\_MIPS\_ISA4\_\_ 162  
\_\_MWBROWSER\_\_ 162  
\_\_MWERKS\_\_ 162  
\_\_option(), プリプロセッサ関数 162  
\_\_powerc 162  
\_\_POWERPC\_\_ 162  
\_\_PreInit\_\_() 51  
\_\_profile\_\_ 162  
\_\_rol() 45  
\_\_ror() 45  
\_\_STDC\_\_ 160  
\_\_stdcall キーワード 37

\_\_TIME\_\_ 160

## 数字

3D 110

## A

a6frames プラグマ 73  
Activate C++ Compiler  
    C/C++ Language 設定パネル 16  
align\_array\_members プラグマ 75  
align プラグマ 74  
altivec\_codegen pragma 76  
altivec\_model pragma 76  
altivec\_vrsave pragma 76  
always\_inline プラグマ 77  
AMD K6 100  
AMD K6 3D 110  
ANSI Keywords Only  
    C/C++ Language 設定パネル 17  
ANSI Strict  
    C/C++ Language 設定パネル 17  
ANSI\_strict プラグマ 77  
ANSI に厳格に従う 33  
ANSI キーワードのみ 36  
arg\_dep\_lookup プラグマ 78  
ARM Conformance  
    C/C++ Language 設定パネル 16  
ARM\_conform 55  
ARM\_conform プラグマ 55, 78  
ARM に適合 54  
asm キーワード 36  
auto\_inline pragma 39  
auto\_inline プラグマ 39, 79  
Auto-inline  
    C/C++ Language 設定パネル 17

## B

bit 回転 45  
bool キーワード 49  
bool プラグマ 80  
bool 型サポート 55

by #pragma segment オプション 140

## C

### C

GNU 拡張 45

C++ コンパイラを必ず利用 54

C++ 例外処理有効 55

C/C++ Language 設定パネル 15

Activate C++ Compiler 16

ANSI Keywords Only 17

ANSI Strict 17

ARM Conformance 16

Auto-inline 17

Deferred Inlinings 17

Don't Reuse Strings 17

EC++ Compatibility Mode 17

Enable bool Support 17

Enable C++ Exceptions 16

Enable Objective C 17

Enable RTTI 16

Enable wchar\_t Support 17

Expand Trigraphs 17

Inline Depth 17

Map Newlines to CR 17

Multi-Byte Aware 17

Pool Strings 17

Relaxed Pointer Type Rules 17

Require Function Prototypes 17

Use Unsigned Chars 17

C/C++ Warning 設定パネル 18

Empty Declararions 19

Extended Error Checking 19

Extra Commas 19

Hidden Virtual Functions 19

Illegal Pragmas 19

Implicit Arithmetic Conversions 19

Inconsitent Use of 'class' and 'struct' Keywords  
19

Non-Inlined Functions 19

Possible Errors 19

Treat All Waarnings As Errors 19

Unused Arguments 19

Unused Variables 19

C/C++ 警告設定パネル 18

C/C++ 言語設定パネル 15

catch ステートメント 47, 55, 57, 94

char 43

check\_header\_flags プラグマ 80

CIncludes 30

code\_seg プラグマ 81

code68020 プラグマ 81

code68881 プラグマ 82

const\_cast キーワード 49

cplusplus プラグマ 54, 82

cpp\_extensions プラグマ 56, 83

CR の代わりに NL を利用 42

## D

d0\_pointers プラグマ 84

def\_inherited プラグマ 53, 85

defer\_codegen プラグマ 86

Deferred Inlinings

C/C++ Language 設定パネル 17

direct\_to\_som プラグマ 90

disable\_registers プラグマ 91

DLL 25

dollar\_identifiers プラグマ 91

Don't Reuse Strings

C/C++ Language 設定パネル 17

dont\_inline プラグマ 38, 92

dont\_reuse\_strings プラグマ 40, 92

dynamic\_cast オペレータ 57

dynamic\_cast キーワード 49, 133

-d オプション 30

D 定数接尾子 45

## E

EC++ Compatibility Mode

C/C++ Language 設定パネル 17

ecplusplus プラグマ 93

EIPC\_EIPSW プラグマ 93

#else 35

Empty Declararions

C/C++ Warning 設定パネル 19

Enable bool Support

C/C++ Language 設定パネル 17

Enable C++ Exceptions

C/C++ Language 設定パネル 16

Enable Objective C

C/C++ Language 設定パネル 17

Enable RTTI

C/C++ Language 設定パネル 16

Enable wchar\_t Support

C/C++ Language 設定パネル 17

#endif 35

enumsalwaysint プラグマ 94

enum は常に Int 型 31

enum 型 23, 31

=

意図しない 20

=

オペレータ 50

exceptions プラグマ 94

Expand Trigraphs

C/C++ Language 設定パネル 17

explicit キーワード 49

Export Symbols オプション 105

export プラグマ 95

.exp ファイル 105

Extended Error Checking

C/C++ Warning 設定パネル 19

extended\_errorchecking プラグマ 24, 96

Extra Commas

C/C++ Warning 設定パネル 19

## F

false キーワード 49

far\_code プラグマ 98

far\_data プラグマ 98

far\_strings プラグマ 99

far\_vtables プラグマ 99

far キーワード 36

flot\_constants プラグマ 100

force\_active プラグマ 100

for ステートメント 21, 54

fourbyteints プラグマ 101

fp\_contract プラグマ 101

friend キーワード 48

fullpath\_prepdump pragma 102

## G

gcc\_extensions pragma 103

global\_optimizer プラグマ 103

GNU C 45

## H

header files 102

Hidden Virtual Functions

C/C++ Warning 設定パネル 19

## I

IEEEdoubles プラグマ 104

if ステートメント 21, 54

ignore\_oldstyle プラグマ 105

Illegal Pragmas

C/C++ Warning 設定パネル 19

Implicit Arithmetic Conversions

C/C++ Warning 設定パネル 19

import プラグマ 105

Inconsistent Use of 'class' and 'struct' Keywords

C/C++ Warning 設定パネル 19

inherited キーワード 52, 85

init\_seg プラグマ 106

Inline Depth

C/C++ Language 設定パネル 17

inline\_depth プラグマ 107

inline\_intrinsics プラグマ 107

inline キーワード 37

Intel MMX 110, 115

internal プラグマ 108

interrupt pragma 109

interrupt\_fast pragma 109

## K

K6 3D 110

## L

lib\_export プラグマ 110

line\_prepdump pragma 111

long long 43, 112

longlong\_enums プラグマ 112

longlong\_prepval プラグマ 112

longlong プラグマ 111

## M

macsbug プラグマ 113

main() 48

Map Newlines to CR

C/C++ Language 設定パネル 17  
message プラグマ 114  
microsoft\_exceptions プラグマ 114  
microsoft\_RTTI プラグマ 114  
MMX 110, 115  
mmx プラグマ 115  
mpwc\_newline プラグマ 43, 117  
mpwc\_relax プラグマ 43, 117  
mpwc プラグマ 116  
Multi-Byte Aware  
    C/C++ Language 設定パネル 17  
MultiMedia 拡張 110, 115  
mutable キーワード 49

## N

namespace キーワード 49  
near\_code プラグマ 98  
no\_static\_dtors プラグマ 119  
Non-Inlined Functions  
    C/C++ Warning 設定パネル 19

## O

oldstyle\_symbols プラグマ 113  
once プラグマ 119  
only\_std\_keywords プラグマ 37, 120  
operator= 50  
opt\_common\_subs プラグマ 120  
opt\_dead\_assignments プラグマ 120  
opt\_dead\_code プラグマ 121  
opt\_lifetimes プラグマ 121  
opt\_loop\_invariants プラグマ 122  
opt\_proagation プラグマ 122  
opt\_strength\_reduction プラグマ 122  
opt\_unroll\_loops プラグマ 123  
opt\_vectorize\_loops プラグマ 123  
optimization\_level プラグマ 103  
optimize\_for\_size プラグマ 124  
\_\_option(), プリプロセッサ関数 162  
options align= プラグマ 74

## P

pack プラグマ 124  
parameter プラグマ 125

pascal キーワード 37  
pcrelstrings プラグマ 126  
peephole プラグマ 126  
pixel 37  
pointers\_in\_A0 プラグマ 128  
pointers\_in\_D0 プラグマ 128  
Pool Strings  
    C/C++ Language 設定パネル 17  
pool\_data プラグマ 129  
pool\_strings プラグマ 40, 129  
pop プラグマ 130  
Possible Errors  
    C/C++ Warning 設定パネル 19  
PowerPC  
    VRSave 76  
ppc\_unroll\_factor\_limit pragma 127  
ppc\_unroll\_instructions\_limit pragma  
    127  
ppc\_unroll\_speculative pragma 127  
#pragma ステートメント 72  
precompile\_target プラグマ 131  
\_\_PrelInit\_\_() 51  
preprocessor  
    #line directive 111  
    header files 102  
profile プラグマ 131  
push プラグマ 130

## Q

qualified name syntax 62

## R

readonly\_strings プラグマ 132  
register 134  
reinterpret\_char キーワード 49  
Relaxed Pointer Type Rules  
    C/C++ Language 設定パネル 17  
Require Function Prototypes  
    C/C++ Language 設定パネル 17  
require\_prototypes プラグマ 42, 132, 133  
return ステートメント  
    欠落した 23  
return ステートメント  
    空の 23

RTTI プラグマ 133  
 RTTI 有効 55  
 Run-Time Type Information 55, 133

## S

`schedule pragma` 134  
`scheduling プラグマ` 134  
 Section Mappings 設定パネル 90  
`section プラグマ` 135  
`segment プラグマ` 140  
`side_effects プラグマ` 140  
`simple_prepdump プラグマ` 141  
`size_t` 30  
`sizeof()` オペレータ 30  
`smart_code プラグマ` 98  
 SOM Call Optimization プラグマ 141  
 SOMCallStyle プラグマ 142  
 SOMCheckEnvironment プラグマ 143  
 SOMClassVersion プラグマ 144  
 SOMMetaClass プラグマ 144  
 SOMReleaseOrder プラグマ 145  
`stack_cleanup プラグマ` 146  
`static_cast` キーワード 49  
`static_inlines プラグマ` 146  
`suppress_init_code プラグマ` 147  
`switch` ステートメント 54  
`sym プラグマ` 147  
`syspath プラグマ` 147

## T

`template class` ステートメント 63  
`toc_data プラグマ` 148  
`traceback プラグマ` 149  
 Treat All Warnings As Errors  
   C/C++ Warning 設定パネル 19  
`trigraphs プラグマ` 37, 148  
 Trigraphs 拡張 37  
 Trigraph 拡張 37  
 Trigraph 文字 37  
`true` キーワード 49  
`try` ステートメント 47, 55, 57, 94  
`type_info` 59  
`typeid` キーワード 49, 133

`typename` 62

## U

Unicode 39  
`unsigned char` 43  
 Unsigned Char を使用 43  
`unsigned_char プラグマ` 149  
 Unused Arguments  
   C/C++ Warning 設定パネル 19  
 Unused Variables  
   C/C++ Warning 設定パネル 19  
`unused プラグマ` 21, 22, 150  
 Use Unsigned Chars  
   C/C++ Language 設定パネル 17  
`use_fp_instructions プラグマ` 150  
`use_frame プラグマ` 151  
`use_mask_registers プラグマ` 151  
`using` キーワード 49

## V

`vector` 37  
`virtual` キーワード 48  
 VRSave レジスタ 76

## W

`warn_emptydecl プラグマ` 20, 151  
`warn_extracomma プラグマ` 23, 152  
`warn_hidevirtual プラグマ` 153  
`warn_illpragma プラグマ` 20, 153  
`warn_padding プラグマ` 155  
`warn_possunwant プラグマ` 21, 156  
`warn_resultnotused プラグマ` 157  
`warn_structclass プラグマ` 157  
`warn_unusedarg プラグマ` 22, 158  
`warn_unusedvar プラグマ` 22, 158  
`warning_errors プラグマ` 20, 152  
`warning プラグマ` 159  
`wchar_type プラグマ` 159  
`wchar_t` 型サポート 34  
`while` ステートメント 21, 54

## ア行

暗黙の算術変換 24

インクルードファイル 28

インスタンス化

    テンプレート 62

インライン 86

インラインの深さ 38

インライン化しない 38

エラーと警告ウィンドウ

    IDE 114

オプションのチェック 162

## 力行

拡張エラーチェック 23

仮関数が隠ぺいされた場合 24

必ずユーザーパスを検索する 29

漢字 39

関数がインライン展開されなかった場合 25

関数結果の警告 157

関数プロトタイプが必要 41

キーワード 'class' と 'struct' の一貫性のない使用 25

基底クラス

    プロテクトされた 54

キャラクタ, マルチバイト 37

グローバルデストラクタ 119

警告

    エラー 20

    定義 18

警告をエラーとしてあつかう 20

構造体

    匿名 56

コピーコンストラクタ 50

コマンドライン 30

コメント、C++ 形式の 34

## サ行

識別子 28

    最大長 28

識別子がない宣言 20

条件判断オペレータ 54

スマート 38

整数フォーマット 43

静的デストラクタ 119

接尾子, 定数 45

宣言

    ステートメントで変数を 54

テンプレート 60

## タ行

タイプチェック 43

単純なクラス 50

代入, 意図しない 20

定義前のインライン展開を許可 39, 86

テンプレート 60

    インスタンス化 62

    宣言 60

デストラクタ 119

匿名の構造体 56

## ハ行

引数

    名前のない 34

ファイルマッピング言語設定パネル 54

不当な pragma 20

プラグマ

    全リスト 71

    有効範囲 72

プリフィックスファイル 30

プリプロセッサ

    long long 式 112

    と # 35

プロトタイプ 41

ヘッダファイル 28

変数

    volatile 30

    アドレスを指定 36

ポインタタイプルールを緩める 43

## マ行

マクロ

    and # 35

間違い易いエラー 20

マルチバイトキャラクタ 37

マルチバイト文字 (日本語など) に対応 39

マングル名 28

未使用引数 22

未使用変数 21

無限ループ 21

    作成 21

メンバー関数へのポインタ 56

---

文字列定数

    プール 39

文字列定数を一カ所にまとめる 39

文字列定数を再利用しない 40

文字列リテラル

    再使用 40

## ヤ行

余分なコンマ 23

## ラ行

ライブラリ 25

リンク

    識別子の長さ 28

例外処理 55

列挙型 31





# CodeWarrior

## C Compilers Reference

### Credits

writing lead: Marc Paquette

other writers: Mark Anderson, Gene Backlin,  
BitHead, Jeff Mattson, Jim Trudeau

engineering: Mark Anderson, Bob Campbell,  
Ben Combee, Pascal Cleve, Rajeev Gulati,  
Andreas Hommel, Udi Kalekin, Michael Kahl,  
Bob Kushlis, John McEnerney, Fred Peterson,  
Laurent Visconti, Rhonda Wittels

frontline warriors: Richard Atwell, John C. Daub, Ron Liechty,  
John Roseborough, Joel Sumner,  
Jim Trudeau, L. Frank Turovich,  
CodeWarrior users everywhere

translation: Naomi Owashi

proofreader: Chizu Kanbara

## CodeWarrior 文書のガイド

CodeWarrior の文書はツールと同様にモジュールのように構成されています。ツール、言語、ライブラリ、ターゲットごとにマニュアルがあります。各 CodeWarrior 製品によって含まれるマニュアルが異なります。この表に記載されていないマニュアルが含まれることもあります。

コアマニュアル	
IDE User Guide	CodeWarrior IDE と CodeWarrior デバッガの使い方
言語 / コンパイラのマニュアル	
C Compilers Reference	C/C++ フロントエンドコンパイラの情報
Error Reference	コンパイラ / リンカのエラーメッセージのリストおよび解説
Assembler Reference	スタンドアローンアセンブラのシンタックス
Command-Line Tools Reference	Mac OS MPW コンパイラのコマンドラインのオプション
Plugin API Manual	CodeWarrior のプラグインコンパイラ / リンカの API
ライブラリのマニュアル	
MSL C Reference	Metrowerks ANSI 標準 C ライブラリの関数のリファレンス
MSL C++ Reference	Metrowerks ANSI 標準 C++ ライブラリの関数のリファレンス
MFC Reference	Win32 用の Microsoft Foundation Classes リファレンス
Win32 SDK Reference	Win32 API の Microsoft リファレンス
The PowerPlant Book	Mac OS 用アプリケーションフレームワークのガイド
PowerPlant Advanced Topics	PowerPlant での Mac OS プログラミングの高度なテクニック
ターゲットマニュアル	
Targeting Java VM	Java 仮想マシンプログラミングでの CodeWarrior の使い方
Targeting Mac OS	Mac OS プログラミングでの CodeWarrior の使い方
Targeting MIPS	MIPS 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting NEC V800/V850 series	NEC V810/830 プロセッサプログラミングでの CodeWarrior の使い方
Targeting Net Yaroze	Net Yaroze プログラミングでの CodeWarrior の使い方
Targeting PlayStation	PlayStation プログラミングでの CodeWarrior の使い方
Targeting PlayStation2	PlayStation2 プログラミングでの CodeWarrior の使い方
Targeting PowerPC	PPC 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting Windows	Windows プログラミングでの CodeWarrior の使い方

印の付いているマニュアルは日本語訳が用意されています。