

Tru64 UNIX オペレーティング・システム

Compaq C 言語リファレンス・マニュアル

AA-RK3VE-TE

2002 年 11 月

本書は Compaq C プログラミング言語のリファレンス・マニュアルです。

ソフトウェア・バージョン: Compaq C for Tru64 UNIX V5.1B 以降
 Compaq C V6.5 for OpenVMS システム

日本ヒューレット・パッカード株式会社

2002 年 11 月

© 2002 日本ヒューレット・パッカート株式会社

本書の著作権は日本ヒューレット・パッカート株式会社が保有しており、本書中の解説および図、表は日本ヒューレット・パッカートの文書による許可なしに、その全体または一部を、いかなる場合にも再版あるいは複製することを禁じます。

また、本書に記載されている事項は、予告なく変更されることがありますので、あらかじめご承知おきください。万一、本書の記述に誤りがあった場合でも、日本ヒューレット・パッカートは一切その責任を負いかねます。

本書で解説するソフトウェア (対象ソフトウェア) は、所定のライセンス契約が締結された場合に限り、その使用あるいは複製が許可されます。

COMPAQ, Compaq ロゴ, VAX, Alpha, VMS, OpenVMS, および Tru64 は、アメリカ合衆国ならびに他の国における Compaq Information Technologies Group, L.P. の商標です。UNIX は、アメリカ合衆国ならびに他の国における The Open Group の商標です。本書に記載されているその他の製品名は、各社の商標です。

本書は、日本語 VAX DOCUMENT V 2.1 を用いて作成しています。

目次

まえがき	xv
1 C 言語の要素	
1.1 文字集合	1-3
1.1.1 3 文字表記	1-6
1.1.2 2 文字表記	1-7
1.2 識別子	1-7
1.3 ユニバーサル・キャラクタ名	1-9
1.4 コメント	1-9
1.5 キーワード	1-11
1.6 演算子	1-13
1.7 区切り記号	1-14
1.8 文字列リテラル	1-15
1.9 定数	1-16
1.9.1 整数定数	1-17
1.9.2 浮動小数点定数	1-20
1.9.2.1 意味	1-20
1.9.2.2 浮動小数点型	1-21
1.9.2.3 16 進浮動小数点定数	1-21
1.9.2.4 例	1-22
1.9.3 文字定数	1-22
1.9.3.1 ワイド文字	1-23
1.9.3.2 多バイト文字	1-24
1.9.3.3 文字エスケープ・シーケンス	1-25
1.9.3.4 数値エスケープ・シーケンス	1-26
1.9.4 列挙定数	1-27
1.10 ヘッダ・ファイル	1-27
1.11 制限事項	1-28

1.11.1	翻訳上の制限	1-28
1.11.2	数値制限	1-29
1.11.3	文字表示	1-30
2	基本事項	
2.1	ブロック	2-2
2.2	コンパイル単位	2-3
2.3	スコープ	2-4
2.3.1	ファイル・スコープ	2-4
2.3.2	ブロック・スコープ	2-5
2.3.3	関数スコープ	2-6
2.3.4	関数プロトタイプ・スコープ	2-6
2.4	可視性	2-7
2.5	副作用と評価順序点	2-8
2.6	不完全型	2-9
2.7	互換型と合成型	2-10
2.8	結合	2-13
2.9	仮定義	2-15
2.10	記憶域クラス	2-15
2.10.1	auto クラス	2-16
2.10.2	register クラス	2-17
2.10.3	static クラス	2-18
2.10.4	extern クラス	2-18
2.11	記憶域クラス修飾子	2-18
2.11.1	__inline 修飾子	2-20
2.11.2	inline 修飾子	2-20
2.11.2.1	例 — インライン関数指定子の使用	2-22
2.11.3	__forceinline 修飾子	2-25
2.11.4	__align 修飾子	2-25
2.12	順方向参照	2-27
2.13	タグ	2-28
2.14	左辺値と右辺値	2-30
2.15	名前空間	2-31
2.16	前処理	2-32

2.17	型名	2-33
3	データ型	
3.1	データ・サイズ	3-5
3.2	汎整数型	3-6
3.2.1	非文字型	3-7
3.2.2	文字型	3-9
3.3	浮動小数点型	3-11
3.3.1	複素数型 (<i>Alpha only</i>)	3-11
3.3.2	虚数型 (<i>Alpha only</i>)	3-12
3.4	派生型	3-13
3.4.1	関数型	3-13
3.4.2	ポインタ型	3-14
3.4.3	配列型	3-15
3.4.4	構造体型	3-16
3.4.5	共用体型	3-18
3.5	void 型	3-20
3.6	列挙型	3-22
3.7	型修飾子	3-23
3.7.1	const 型修飾子	3-24
3.7.2	volatile 型修飾子	3-26
3.7.3	__unaligned 型修飾子	3-28
3.7.4	__restrict 型修飾子	3-29
3.7.4.1	理由	3-29
3.7.4.1.1	エイリアシング	3-30
3.7.4.1.2	ライブラリの例	3-30
3.7.4.1.3	相互に重なり合うオブジェクト	3-31
3.7.4.1.4	memcpy のための限定ポインタのプロ トタイプ	3-33
3.7.4.2	__restrict 型修飾子の正式な定義	3-33
3.7.4.3	例	3-35
3.7.4.3.1	ファイル・スコープ限定ポインタ	3-35
3.7.4.3.2	関数の仮引数	3-36
3.7.4.3.3	ブロック・スコープ	3-37
3.7.4.3.4	構造体のメンバ	3-38
3.7.4.3.5	型定義	3-38
3.7.4.3.6	限定ポインタに基づく式	3-38
3.7.4.3.7	限定ポインタ間での代入	3-40
3.7.4.3.8	非限定ポインタへの代入	3-41

	3.7.4.3.9	型修飾子の非効率的な使用	3-42
	3.7.4.3.10	制約違反	3-42
3.8	型定義		3-43
4	宣言		
4.1	宣言の構文規則		4-2
4.2	初期化		4-5
4.3	外部宣言		4-8
4.4	単純オブジェクトの宣言		4-11
	4.4.1	単純オブジェクトの初期化	4-11
	4.4.2	整数オブジェクトの宣言	4-12
	4.4.3	文字変数の宣言	4-13
	4.4.4	浮動小数点変数の宣言	4-13
4.5	列挙型の宣言		4-13
4.6	ポインタの宣言		4-15
	4.6.1	void ポインタの宣言	4-17
	4.6.2	ポインタの初期化	4-18
4.7	配列の宣言		4-19
	4.7.1	配列の初期化	4-22
	4.7.2	ポインタと配列	4-25
	4.7.3	可変長配列	4-26
4.8	構造体と共用体の宣言		4-28
	4.8.1	構造体と共用体の類似点	4-32
	4.8.2	構造体と共用体の相違点	4-32
	4.8.3	ビット・フィールド	4-34
	4.8.4	構造体の初期化	4-36
	4.8.5	共用体の初期化	4-38
4.9	ディジグネーションを使用した初期化子		4-39
	4.9.1	カレント・オブジェクト	4-39
	4.9.2	ディジグネーション	4-40
	4.9.3	例	4-41
4.10	タグの宣言		4-43
4.11	型定義の宣言		4-44

5	関数	
5.1	関数呼出し	5-1
5.2	関数型	5-2
5.3	関数定義	5-3
5.4	関数宣言	5-7
5.5	関数プロトタイプ	5-9
5.5.1	プロトタイプの構文	5-10
5.5.2	スコープと変換	5-12
5.6	仮引数と実引数	5-13
5.6.1	実引数の変換	5-14
5.6.2	実引数としての関数識別子と配列識別子	5-15
5.6.3	main 関数への実引数の引渡し	5-16
6	式と演算子	
6.1	1 次式	6-2
6.1.1	識別子	6-2
6.1.2	定数	6-2
6.1.3	文字列リテラル	6-3
6.1.4	括弧で囲んだ式	6-3
6.2	C 言語演算子	6-3
6.3	後置演算子	6-8
6.3.1	配列の参照	6-8
6.3.2	関数呼出し	6-10
6.3.3	構造体と共用体の参照	6-12
6.3.4	後置インクリメント演算子と後置デクリメント演算子	6-13
6.4	単項演算子	6-15
6.4.1	単項プラス演算子と単項マイナス演算子	6-15
6.4.2	否定	6-16
6.4.3	前置インクリメント演算子と前置デクリメント演算子	6-16
6.4.4	アドレス演算子と間接参照	6-18
6.4.5	ビット否定	6-19
6.4.6	キャスト演算子	6-19
6.4.7	sizeof 演算子	6-21
6.4.8	__typeof__ 演算子	6-22
6.4.9	_Pragma 演算子	6-23
6.5	2 項演算子	6-24
6.5.1	乗除演算子	6-25

6.5.2	加減演算子	6-26
6.5.3	シフト演算子	6-27
6.5.4	関係演算子	6-28
6.5.5	等価演算子	6-29
6.5.6	ビット演算子	6-30
6.5.7	論理演算子	6-30
6.6	条件演算子	6-31
6.7	代入演算子	6-32
6.8	コンマ演算子	6-34
6.9	定数式	6-34
6.9.1	汎整数定数式	6-35
6.9.2	算術定数式	6-35
6.9.3	アドレス定数	6-36
6.10	複合リテラル式	6-36
6.11	データ型変換	6-40
6.11.1	通常の算術変換	6-41
6.11.1.1	文字と整数	6-42
6.11.1.2	符号付き整数と符号なし整数	6-43
6.11.1.3	浮動小数点と汎整数	6-44
6.11.1.4	浮動小数点型	6-44
6.11.2	ポインタ変換	6-45
6.11.3	関数の実引数の変換	6-45
7	文	
7.1	ラベル付き文	7-1
7.2	複文	7-2
7.3	式文	7-3
7.4	空文	7-3
7.5	選択文	7-4
7.5.1	if 文	7-4
7.5.2	switch 文	7-5
7.6	繰返し文	7-8
7.6.1	while 文	7-9
7.6.2	do 文	7-9
7.6.3	for 文	7-10
7.7	飛越し文	7-11

7.7.1	goto 文	7-12
7.7.2	continue 文	7-12
7.7.3	break 文	7-13
7.7.4	return 文	7-13

8 前処理命令とあらかじめ定義されたマクロ

8.1	マクロ定義 (#define および #undef)	8-2
8.1.1	オブジェクト形式マクロ	8-4
8.1.2	関数形式マクロ	8-5
8.1.2.1	マクロ定義指定の規則	8-6
8.1.2.2	マクロ参照指定の規則	8-6
8.1.2.3	マクロ実引数における副作用	8-7
8.1.3	文字列リテラルへの変換 (#)	8-7
8.1.4	トークンの連結 (##)	8-9
8.2	条件付きコンパイル	
	(#if, #ifdef, #ifndef, #else, #elif, #endif, defined)	8-9
8.2.1	#if 命令	8-10
8.2.2	#ifdef 命令	8-11
8.2.3	#ifndef 命令	8-11
8.2.4	#else 命令	8-11
8.2.5	#elif 命令	8-11
8.2.6	#endif 命令	8-12
8.2.7	defined 演算子	8-12
8.3	ファイルの取込み (#include)	8-13
8.4	明示的な行番号付け (#line)	8-14
8.5	処理系固有の前処理命令 (#pragma)	8-15
8.6	エラー命令 (#error)	8-19
8.7	空命令 (#)	8-19
8.8	あらかじめ定義されたマクロ名	8-19
8.8.1	__DATE__ マクロ	8-19
8.8.2	__FILE__ マクロ	8-20
8.8.3	__LINE__ マクロ	8-20
8.8.4	__TIME__ マクロ	8-20
8.8.5	__STDC__ マクロ	8-20
8.8.6	__STDC_HOSTED__ マクロ	8-21
8.8.7	__STDC_VERSION__ マクロ	8-21
8.8.8	__STDC_ISO_10646__ マクロ	8-21
8.8.9	システム識別マクロ	8-21

8.9	<code>__func__</code> 宣言済み識別子	8-22
9	ANSI C 標準ライブラリ	
9.1	診断メッセージ (<code><assert.h></code>)	9-2
9.2	複素数算術 (<code><complex.h></code>)	9-3
9.3	文字処理 (<code><ctype.h></code>)	9-10
9.4	エラー・コード (<code><errno.h></code>)	9-12
9.5	ANSI C の限界 (<code><limits.h></code> および <code><float.h></code>)	9-13
9.6	ローカル化 (<code><locale.h></code>)	9-13
9.7	算術 (<code><math.h></code>)	9-19
9.8	非ローカル飛越し (<code><setjmp.h></code>)	9-23
9.9	シグナル処理 (<code><signal.h></code>)	9-23
9.10	可変個実引数 (<code><stdarg.h></code>)	9-25
9.11	ブール型とブール値 (<code><stdbool.h></code>)	9-26
9.12	共通定義 (<code><stddef.h></code>)	9-27
9.13	標準入出力 (<code><stdio.h></code>)	9-28
9.14	汎用ユーティリティ (<code><stdlib.h></code>)	9-41
9.15	文字列処理 (<code><string.h></code>)	9-50
9.16	型汎用数学 (<code><tgmath.h></code>)	9-54
9.16.1	実数型の決定	9-55
9.16.2	<code><math.h></code> と <code><complex.h></code> で同じ名前の接尾語なしの関数	9-55
9.16.3	対応する c 接頭語関数が <code><complex.h></code> にない, <code><math.h></code> の接尾語なし関数	9-56
9.16.4	<code><math.h></code> 内の関数に対応する c 接頭語付き関数ではない, <code><complex.h></code> 内の接尾語なし関数	9-56
9.16.5	例	9-57
9.16.6	虚数引数	9-58
9.17	日付と時刻 (<code><time.h></code>)	9-58

A 言語構文一覧

B ANSI 規格準拠の概要

B.1	診断 (§2.1.1.3)	B-2
B.2	ホスト環境 (§2.1.2.2)	B-2
B.3	多バイト文字 (§2.2.1.2)	B-2
B.4	エスケープ・シーケンス (§2.2.2)	B-2
B.5	翻訳限界 (§2.2.4.1)	B-3
B.6	数量的限界 (§2.2.4.2)	B-3
B.7	キーワード (§3.1.1)	B-4
B.8	識別子 (§3.1.2)	B-5
B.9	識別子の結合 (§3.1.2.2)	B-5
B.10	型 (§3.1.2.5)	B-6
B.11	整数定数 (§3.1.3.2)	B-6
B.12	文字定数 (§3.1.3.4)	B-6
B.13	文字列リテラル (§3.1.4)	B-6
B.14	演算子 — 複合代入 (§3.1.5)	B-7
B.15	文字と整数 — 値保存拡張 (§3.2.1.1)	B-7
B.16	符号付き整数と符号なし整数の変換 (§3.2.1.2)	B-8
B.17	浮動小数点数と汎整数の変換 (§3.2.1.3)	B-8
B.18	ポインタ変換 (§3.2.2.3)	B-9
B.19	構造体および共用体メンバ (§3.3.2.3)	B-9
B.20	sizeof 演算子 (§3.3.3.4)	B-9
B.21	キャスト演算子 (§3.3.4)	B-9
B.22	乗除演算子 (§3.3.5)	B-10
B.23	加減演算子 (§3.3.6)	B-10
B.24	ビット単位のシフト演算子 (§3.3.7)	B-11
B.25	記憶域クラス指定子 (§3.5.1)	B-11
B.26	型指定子 (§3.5.2)	B-11

B.27	構造体および共用体指定子 (§3.5.2.1)	B-11
B.28	変形構造体および共用体	B-12
B.29	構造体の境界調整	B-12
B.30	列挙型指定子 (§3.5.2.2)	B-14
B.31	型修飾子 (§3.5.3)	B-14
B.32	宣言子 (§3.5.4)	B-14
B.33	初期化 (§3.5.7)	B-15
B.34	switch 文 (§3.6.4.2)	B-15
B.35	外部オブジェクト定義 (§3.7.2)	B-15
B.36	条件付き取込み (§3.8.1)	B-15
B.37	ソース・ファイルの取込み (§3.8.2)	B-16
B.38	マクロ置換 — あらかじめ定義されたマクロ名 (§3.8.3)	B-16
B.39	##演算子 (§3.8.3.3)	B-19
B.40	エラー命令 (§3.8.5)	B-19
B.41	プラグマ命令 (§3.8.6)	B-20
B.42	関数のインライン展開	B-20
B.43	結合プラグマ	B-21
B.44	その他のプラグマ	B-21

C ASCII 等価テーブル

D Compaq C でサポートするコモン C の拡張

D.1	ANSI C と互換性のある拡張	D-1
D.2	ANSI C と互換性のない拡張	D-3

E Compaq C でサポートするVAX Cの拡張

E.1	ANSI C と互換性のある拡張	E-1
E.2	ANSI C と互換性のない拡張	E-4

F 識別子のユニバーサル・キャラクタ名

索引

例

4-1	構造体の初期化の規則	4-37
5-1	実引数として渡される関数の宣言	5-15
7-1	switch 文を使用した空白, タブ, および改行のカウント	7-7

表

1-1	3 文字表記	1-6
1-2	2 文字表記	1-7
1-3	キーワード	1-11
1-4	区切り記号	1-14
1-5	整数定数に型を割り当てる規則	1-18
1-6	浮動小数点の表記	1-22
1-7	文字エスケープ・シーケンス	1-25
2-1	定義済み境界調整定数	2-26
2-2	型名の例	2-33
3-1	基本データ型	3-3
3-2	データ型のサイズと範囲	3-5
6-1	C 言語演算子	6-3
6-2	C 言語演算子の優先順位	6-6
9-1	ファイル・モード	9-32
9-2	strftime 変換指定子	9-60
B-1	Tru64 UNIX 上であらかじめ定義されたマクロ名	B-17
B-2	OpenVMS VAX および Alpha 上であらかじめ定義されたマクロ名...	B-17
B-3	ライブラリ・ルーチン規格準拠マクロ — 全プラットフォーム	B-19

C-1	ASCII 等価チャート	C-1
-----	--------------------	-----

まえがき

本書は、弊社のシステム上で *Compaq C* プログラミング言語を使用するための参照情報について説明します。*Compaq C* は、*OpenVMS VAX* システム、*OpenVMS Alpha* システム、および *Tru64 UNIX* システム用の ISO/ANSI 規格に準拠した C コンパイラです。*Tru64 UNIX* は Alpha プロセッサ上で動作します。

Compaq C は ISO (国際標準化機構) の C 規格 (ISO 9899:1990[1992])、以前の『American National Standard for Information Systems-Programming Language C』(ドキュメント番号: X3.159-1989) に準拠しています。コマンド行オプションを使用することにより、コモン仕様の C 言語 (Kernighan と Ritchie の C 言語) および VAX C を含む旧形式の C 言語との適合性を有する *Compaq C* を使用することができます。

本書は、American National Standard による C プログラミング言語用の Information Systems X3J11 委員会の規格 (本書では ANSI C 規格と呼ぶ) に基づいています¹。本書では、すべてのライブラリ関数と ANSI C 規格に対する C 言語機能拡張についても説明しています。

対象読者

本書は、*Compaq C* (以前の DEC C) に関する参照情報を必要とするプログラマーを対象にしています。本書ではタスク指向またはプラットフォーム固有の情報についてはほとんど触れていません。こういった情報については、プラットフォームに固有の *Compaq C* のマニュアル (*OpenVMS* システム用のユーザズ・ガイドおよびオンライン・ヘルプ、または *Tru64 UNIX* システム用のプログラミング・ガイドおよびリファレンス・ページ) を参照してください。

¹ 弊社は American National Standard Programming Language C から文書の全体または一部を引用させていた
だいたことについて、CBEMA および Accredited Standards Committee X3 に謝辞を申し上げます。

ANSI 規格の目的

ANSI C 規格は、C 言語の仕様が明確でないために発生する問題を処理することを目的として、プログラム開発者と C 言語に精通したユーザから構成される委員会によって作成されました。このような問題は主として、異機種のマシン間でのプログラムの移植性に関連するものでした。この委員会では、構文と意味があいまいな領域、または確定していない領域について C 言語を分析し、正確な定義のみを C 言語構造に採用しました。その結果、不明瞭な点がなくなり、マシンに依存しない定義になりました。

ANSI C 規格には、次のように記述されています。

「形式を指定し、C プログラミング言語で表現するプログラムの解釈を確立します。ANSI C 規格の目的は、さまざまなコンピューティング・システム上における C 言語プログラムについての移植性、信頼性、保守性および効果的な実行を促進することです。」

ANSI C 規格では以下の項目について明記しています。

- C 言語の表現、構文および制約
- C プログラムの解釈のための意味規則
- C プログラムでの入出力の表記

ANSI C 規格では以下の項目については明記していません。

- C プログラムのコンパイル方法
- C プログラムのリンク方法
- C プログラムの実行方法
- ANSI C プログラムを実行するマシンのサイズに関する上限と下限

新しい機能と変更された機能

本書は、*Compaq C* バージョン 6.5 の以下の新しい言語機能を反映して改訂されました。

- 本バージョンでは、GEM BL48 バックエンドを使用し、EV7 プロセッサを完全にサポートしています。
- オプションの "_nm" 接尾語を #pragma 名の後ろに追加し、そのプラグマのマクロ展開を抑制することができます。*Compaq C* バージョン 6.4 で導入された "_m" 接尾語と反対です (第 8.5 節)。
- C99 _Pragma 演算子のサポートが追加され、マクロ展開によって #pragma 命令を効果的に生成できるようになりました (第 6.4.9 項)。
- 無限大や NaN の特定の値を表わす C99 定数がサポートされました (IEEE 浮動小数点数を使用し、-common や -vaxc 以外の言語モードでコンパイルするときのみ) (第 9.7 節)。
- C99 の隣接文字列の連結がサポートされました。ワイド文字列と通常の文字列を混在させることができ、その場合、通常の文字列がワイド文字列に拡張され、結果はワイド文字列になります (第 1.8 節)。
- ユニバーサル・キャラクタ名のサポートが追加されました。C99 のユニバーサル・キャラクタ名 (UCN) は、識別子、文字列リテラル、および文字定数と、これらのワイド文字版で 사용할ことができます。
- 新規キーワード NOCTRL と RESTORE_CTRL が、#pragma extern_prefix プリプロセッサ命令に追加されました。これらのキーワードは、コンパイラが省略時の RTL 接頭語をプラグマ命令で指定された名前に適用するかどうかを制御します。

本書の構成

本書は、以下に示す章および付録から構成されています。

第 1 章では、C 言語の要素について説明します。

第 2 章では、C 言語の基本的な概念について説明します。

第 3 章では，C 言語のデータ型および型修飾子について説明します。

第 4 章では，*Compaq C* で使用する識別子の宣言について説明します。この章では，定数，変数，構造体，共用体，ポインタ，および配列の宣言について説明します。

第 5 章では，関数呼出し，関数の宣言，関数定義，関数の仮引数，および関数の実引数について説明します。

第 6 章では，C 言語で構築できる式の種類について説明します。単項演算子，2 項演算子，条件演算子，1 次演算子，および後置演算子を含む C 言語で使用する演算子の有効性についても説明します。

第 7 章では，フロー制御，条件付き実行，ループ設定，および割込みを行う C 文について説明します。

第 8 章では，C 前処理命令およびあらかじめ定義されたマクロの使用目的について説明します。

第 9 章では，ヘッダ・ファイルにより用意された ANSI C 標準ライブラリの関数，マクロ，および型について説明します。

付録 A では，C 言語構成に関するすべての構文をまとめて示します。

付録 B では，ANSI C 規格に対する拡張や例外を含む，*Compaq C* の ANSI 規格に準拠した範囲について説明します。

付録 C では，ASCII 文字集合の各文字に対する ASCII の 8 進数，10 進数，および 16 進数値を記載します。

付録 D では，コモン C 適合オプションを使用して *Compaq C* がサポートするコモン C の言語機能拡張を記載します。

付録 E では，VAX C 適合オプションを使用して *Compaq C* がサポートする VAX C の言語機能拡張を記載します。

関連資料

次は、*Compaq C* のプログラミングを行う際に役に立つ資料です。

- 『*Compaq C* for OpenVMS システム ユーザーズ・ガイド』

本書は、*OpenVMS*オペレーティング・システム上で *Compaq C* プログラムを開発およびデバッグするために必要な情報について説明します。また、*C* プログラムを *OpenVMS*および他のオペレーティング・システム間で移植するための情報以外に、*OpenVMS*システムに固有の *Compaq C* 機能についても記述します。

- 『*Compaq C* Run-Time Library Reference Manual for OpenVMS Systems』

*OpenVMS*で取り込む *C* ライブラリ関数に関する完全な参照情報について説明します。

- cc(1) リファレンス・ページ

このリファレンス・ページでは、*Tru64 UNIX*システム上の *Compaq C* で使用できる *cc* コマンド行オプションについて説明します。

- *Tru64 UNIX*ドキュメント・セット

このドキュメント・セットは、*Tru64 UNIX*オペレーティング・システムとそのユーティリティについて説明します。次のマニュアルが特に役に立ちます。

- 『*Tru64 UNIX*プログラミング・ガイド』

本書では、*Tru64 UNIX*オペレーティング・システム上で *C* プログラムの開発およびデバッグを行う際に必要な情報を含め、*Tru64 UNIX*のプログラミング環境について説明します。本書には、*cc(1)* リファレンス・ページとともに、*Tru64 UNIX*システムに固有な *Compaq C* 機能についての記述もあります。

- 『*Tru64 UNIX* Reference Pages』セクション 2 および 3

*Tru64 UNIX*オペレーティング・システムで取り込む *C* ライブラリ関数に関する完全な参照情報について説明します。

- 『ANSI/ISO/IEC 9899:1999 - Programming Languages - C』

1999 年 12 月に ISO から出版され , 2000 年 4 月に ANSI 規格として採用された , C99 規格。

- 『ISO/IEC 9899:1990-1994 - Programming Languages - C, Amendment 1: Integrity』

ISO C , Amendment 1 と呼ばれているドキュメント。

- 『ISO/IEC 9899:1990[1992] - Programming Languages - C』

このドキュメントは , ISO C と呼ばれている。このドキュメントの規約部分は , ANSI C と呼ばれている 『X3.159-1989, American National Standard for Information Systems - Programming Language C』 と同じです。

- 『American National Standard for Information Systems-Programming Language C』

本書は , X3J11 規格委員会による C 言語の分析結果です。本書は , C 言語に精通したプログラマを対象に ANSI C 言語に関する高度な技術について記述しています。

- 『The C Programming Language, 2nd Edition』²

本書は ANSI 規格が最終的に決定される前に発行された本ですが , 現在でも C 言語のリファレンスとして貴重なものです。

ANSI C の言語機能および拡張機能は 『The C Programming Language, 2nd Edition』 で定義されているよりもさらに増加しているため , *Compaq C* の記述に関しては本書を参照してください。

² Brian W. Kernighan and Dennis M. Ritchie, 『The C Programming Language』 (Englewood Cliffs, New Jersey: Prentice Hall, 1988).

本書で使用する表記法

本書では、以下の表記法を使用します。

表記法	意味
<i>OpenVMS</i> システム	特に断らないかぎり、 <i>OpenVMS VAX</i> および <i>OpenVMS Alpha</i> システムのことを表します。
<code>Return</code>	<code>Return</code> は端末上の Return キーを 1 回押すことを表します。
<code>Ctrl/X</code>	<code>Ctrl/X</code> は、Ctrl キーを押したまま指定の端末文字キー (X) を押すことを表します。
<code>float x; . . . x = 5; option, . . .</code>	垂直方向の省略記号は、プログラムのテキストまたはプログラム出力の一部を省略していることを表します。したがって、例中では記述内容に関連した部分しか表示されません。
<code>syntax_{opt}</code>	オプションの構文要素は、略語 <code>opt</code> を下に書き添えて示します。本書の各節に記載されている構文図は、完全な構文を判断するために付録 A の参照が必要になる場合があります。たとえば、ANSI C 規格の構文は潜在的な代入式として定数を取り込みます。
記憶域クラス指定子: <code>auto static register</code>	構文定義で別の行にある各項目は、相互に排他的な代替項目です。
<code>auto</code> 記憶域クラス ... <code>fprintf</code> 関数 ...	モノスペースでの印字は、言語キーワード、別々にコンパイルした外部関数名とファイル名、構文サマリ、および各例で紹介する変数または識別子の参照を示します。

C 言語の要素

C 言語は他の言語と同様に標準文法や文字集合を使用します。この章では、文法と文字集合から構成される固有の要素を次の各節で説明します。

- 文字集合 (第 1.1 節)
- C 言語の識別子の規則 (第 1.2 節)
- ユニバーサル・キャラクタ名の使用 (第 1.3 節)
- プログラムでのコメントの使用 (第 1.4 節)
- キーワード (第 1.5 節)
- C 言語の演算子の使用 (第 1.6 節)
- 区切り文字の使用 (第 1.7 節)
- プログラムでの文字列の使用 (第 1.8 節)
- 定数値の解釈 (第 1.9 節)
- 別のヘッダ・ファイルまたはモジュールに入っている複数のソース・ファイルに共通な関数宣言やその他の定義の取込み (第 1.10 節)
- ANSI C 規格にプログラムを準拠させる場合の制限 (第 1.11 節)

C コンパイラは、ソース・ファイルからの文字ストリームとしてソース・コードを解釈します。これらの文字は、区切り記号、演算子、識別子、キーワード、文字リテラルまたは定数などのトークンと呼ばれる要素に分類されます。トークンは C 言語における最小の構文要素です。コンパイラは、指定された文字列から可能な限り最長のトークンを形成します。つまり、空白が見つかった場合、または次の文字をトークンの一部にできない場合にトークンは終了します。

C 言語の要素

空白は、空白文字、改行文字、タブ文字、フォーム・フィード文字または垂直タブ文字のいずれかです。コメントも空白とみなされます。第 1.1 節にすべての空白文字の一覧を示します。空白はトークンの区切り記号 (二重引用符で囲まれた文字列内は除く) として使用されます。ただし、それ以外の文字ストリームでは無視され、主に読みやすさのために使用されます。また、空白は前処理命令で意味を持つ場合もあります (第 8 章を参照してください)。

たとえば、次のソース・コード行を例にとります。

```
static int x=0; /* Could also be written "static int x = 0;" */
```

コンパイラは、この行を次のようなトークン (1 行ごとに 1 つ) に分けます。

```
static
int
x
=
0
;
```

コンパイラが入力文字列を処理するとき、トークンを識別し、エラー条件を検索します。コンパイラは、エラーについて次の 3 つの種類を識別できます。

- 字句解析エラー

コンパイラが文字ストリームから正しいトークンを形成できない場合 (不正な文字が使用されている場合など) に起こる。

- 解析 (構文) エラー

正しいトークンを形成できるが、コンパイラがトークンから正しい文を作成できない場合に起こる。たとえば、次の行は不正な区切り文字で初期化子並びを囲んでいる。

```
char x[3] = (1,2,3);
```

- 意味エラー

文法的には正しいが、別の C 言語規則に違反している。たとえば、次の行は浮動小数点値をポインタ型に割り当てようとしている。

```
int *x = 5.7;
```

論理エラーはコンパイラでは識別されません。

C 言語で重要な概念はコンパイル単位という考え方です。このコンパイル単位とは、コンパイラでコンパイルする 1 つまたは複数のファイルを表します。

注意

ANSI C 規格では、コンパイル単位を「翻訳単位」と呼びます。本書では、この翻訳単位をコンパイル単位と等しいものとして扱います。

受入れ可能な最小のコンパイル単位は、1 つの外部定義です。ANSI C 規格は、コンパイル単位に関していくつかの基本的な概念を定義しています。コンパイル単位についての詳細は、第 2.2 節で説明します。

宣言を持たないコンパイル単位は、厳密な ANSI 規格モードを除くすべてのモードで受け入れられますが、コンパイラ警告メッセージが出されます。

1.1 文字集合

文字集合は、ソース・プログラムで使用可能か、またはプログラムの実行時に解釈可能な有効文字を定義します。ソース文字集合は、ソース・テキストに使用できる文字集合です。実行文字集合は、プログラムの実行時に使用できる文字集合です。ソース文字集合は、実行文字集合と必ずしも一致している必要はありません。たとえば、ソース・コード作成のために使用する装置上で、実行文字集合が使用できない場合などです。

文字集合は異なるものがいくつか存在します。たとえば、ある文字集合の文字は ASCII 定義に基づいており、別の文字集合は日本語の漢字を組み込んでいます。コンパイラにとっては、使用している各文字集合の相違は関係ありません。つまり、各文字は固有の値を持っているため、C コンパイラは各文字を異なる整数値として

C 言語の要素

1.1 文字集合

扱います。ASCII 文字集合は 255 文字より少なく、これらの文字は 8 ビット以下で表すことができます。ただし、拡張文字集合の中には多くの文字が存在するので、文字の表記に 8 ビットよりも多くのビットが必要な場合があります。これらの大きい文字を格納するために、`wchar_t` (ワイド文字) と呼ばれる特定の型が作成されています。ワイド文字についての詳細は、第 1.9.3.1 項で説明します。

ANSI と互換性のある C コンパイラのほとんどが、ソース文字集合と実行文字集合の両方に対して次の ASCII 文字を受け入れます。各 ASCII 文字は数値と対応します。付録 C に ASCII 文字とその数値を記載しています。

- 26 文字の小文字 (英字)

`a b c d e f g h i j k l m n o p q r s t u v w x y z`

- 26 文字の大文字 (英字)

`A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`

- 10 個の 10 進数字

`0 1 2 3 4 5 6 7 8 9`

- 30 個のグラフィック文字

`! # % ^ & * () - _ = + ~ ' " : ; ? / | \ { } [] , . < > $`

コンパイラの厳密な ANSI モード・オプションを指定した場合に `$` を使用すると、警告メッセージが表示される。

- 5 つの空白文字

空白	<code>' '</code>
水平タブ	<code>'\t'</code>
フォーム・フィード	<code>'\f'</code>
垂直タブ	<code>'\v'</code>
改行文字	<code>'\n'</code>

文字定数と文字列リテラルの場合には、実行文字集合からの文字を文字または数値エスケープ・シーケンスで表すこともできます。これらのエスケープ・シーケンスについては、第 1.9.3.3 項および第 1.9.3.4 項で説明します。

また，ASCII 実行文字には次の制御文字も含まれます。

- 改行文字 (ソース・ファイルには '`\n`' と表記)
- 警告 (ベル) のトーン ('`\a`')
- バックスペース ('`\b`')
- キャリッジ・リターン ('`\r`')
- ヌル文字 ('`\0`')

ヌル文字はすべてのビットが 0 にセットされているバイト文字またはワイド文字であり，文字列の終わりを示すために使用します。文字列についての詳細は，第 1.8 節で説明します。

改行文字は，読みやすさの向上やプリプロセッサの適切な操作のために，ソース文字ストリームを別の行に分割します。

コンパイラが，端末やウィンドウの幅よりも長い行を 1 つの論理行として解釈することが必要な場合があります。継続する行の終わりにバックスラッシュ文字 (`\`) を付けることによって，1 つの論理行を 2 行以上にわたって入力することを示します。バックスラッシュの直後には，改行文字を続けなければなりません。バックスラッシュは，現在の論理行が次の行に継続されることを意味します。次にその例を示します。

```
#define ERROR_TEXT "Your entry was outside the range of \
0 to 100."
```

コンパイラは，処理中にこの行が 1 つの論理行になるように，バックスラッシュ文字と後続の改行文字を削除します。次にその例を示します。

```
#define ERROR_TEXT "Your entry was outside the range of 0 to 100."
```

長い文字列は，バックスラッシュと改行の行継続機能を使用して複数行にわたって継続できます。しかし，文字列の継続は次の行の最初の位置から開始しなければならない場合があります。このため，プログラムのインデントの構成を破壊しなければならない場合があります。ANSI C 規格では，この問題を避けるために別の文字列継続機能を採用しています。すなわち，空白のみで分割されている 2 つの文字列リテラル

は、1つの論理文字列リテラルを形成するという継続機能です。次にその例を示します。

```
printf ("Your entry was outside the range of "  
       "0 to 100.\n");
```

論理行の最大長は 32,767 文字です。

1.1.1 3 文字表記

C 言語で有効な区切り文字のうち、その一部しか持たない文字集合を使用して C プログラムを記述する際に、ANSI Cではソース・ファイル中に 9 つの 3 文字表記を使用することができます。この 3 文字表記は、コンパイルの最初の段階で単一文字に置き換えられます (コンパイルの段階については、第 2.16 節を参照してください)。表 1-1 に、有効な 3 文字表記とその等価文字を示します。

表 1-1 3 文字表記

3 文字表記	等価文字
??=	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

この表以外の 3 文字表記は認識されません。1 つの疑問符 (?) では 3 文字表記は開始されないため、コンパイルにより等価文字に置換されることもありません。たとえば、次のソース行を例にとります。

```
printf ("Any questions???/n");
```

??/ 表記が置換されると、この行は次のように表されます。

```
printf ("Any questions?\n");
```

1.1.2 2 文字表記

2 文字表記の処理は、ISO C 94モードでコンパイルする場合にサポートされます (OpenVMSシステムでは/STANDARD=ISOC94)。

2 文字表記は、3 文字表記と同様に単一文字に翻訳される文字の組み合わせですが、3 文字表記が内部の文字列リテラルを置き換えるのに対し、2 文字表記は置き換えません。表 1-2 に、有効な 2 文字表記とその等価文字を示します。

表 1-2 2 文字表記

2 文字表記	等価文字
<:	[
:>]
<%	{
%>	}
%:	#
%:%:	##

1.2 識別子

識別子は、次の項目の名前を表す文字シーケンスです。

- 変数
- 関数
- ラベル
- 型定義
- 構造体タグ、列挙型タグ、または共用体タグ
- 構造体メンバ、列挙型メンバ、または共用体メンバ

C 言語の要素

1.2 識別子

- 列挙定数
- マクロ
- マクロ仮引数

識別子には、次の規則が適用されます。

- 識別子は 1 つ以上の英字 (大文字または小文字)、ユニバーサル・キャラクタ名、0 ~ 9 の数字、ドル記号 (\$)、アンダスコア文字 (_) からなる。
\$ 文字を使用すると、厳密な ANSI モードではコンパイラから警告メッセージが表示される。
- 識別子では、大文字と小文字が区別される。たとえば、Test1 識別子は test1 識別子とは異なる。
- 識別子の最初の文字には、数字を使用することはできない。
- 最初の文字には、アンダスコアを使用することはできない。ANSI C 規格では、このような識別子を内部名として予約している。
- 識別子内の各ユニバーサル・キャラクタ名には、ISO/IEC 10646 エンコーディングが付録 F で規定された範囲のいずれかに入る文字を指定する必要がある。
- キーワードは識別子ではない (C のキーワードについては、第 1.5 節を参照のこと)。
- ライブラリ関数名を識別子として使用しないこと。C ライブラリ関数名については、第 9 章を参照のこと。新しい関数に既存のライブラリ関数と同じ名前をつけると、既存のライブラリ関数は使用できなくなる。そうなった場合でも予測した結果が出るかもしれないが、プログラムのメンテナンス時に混乱を生じる可能性がある。
- 識別子は通常、空白、区切り記号、または演算子で区切る。たとえば、次のコードの一部には 4 つの識別子がある。

```
struct employee { int number; char sex; } emp;
```

この例の場合、識別子は employee、number、sex、および emp です。また、struct、int、および char はキーワードです。

外部結合のない識別子の有効文字は、最大で 32,767 文字です。外部結合のある識別子の有効文字は、*Tru64 UNIX*システムでは 1023 文字、*OpenVMS*プラットフォームでは 31 文字です (結合についての詳細は、第 2.8 節を参照してください)。 *OpenVMS*上の外部識別子の場合には、大文字と小文字は区別されません。

有効文字内で異なる識別子は、それぞれ異なる識別子です。複数の識別子が非有効文字内でのみ異なる場合には、それらは同じ識別子とみなされます。

1.3 ユニバーサル・キャラクタ名

ユニバーサル・キャラクタ名を使用すると、その他の文字を指定することができます。ユニバーサル・キャラクタは、識別子、文字定数、文字列リテラルに使用することができます、基本文字セットにない文字を指定するために使用します。

ユニバーサル・キャラクタ名は、`\u`または`\U`で始まり、4 桁または 8 桁の 16 進数が続きます。

ユニバーサル・キャラクタ名`\Unnnnnnnnn`は、8 桁の短識別子 (ISO/IEC 10646 で規定) が `nnnnnnnn` である文字を示します。同様に、ユニバーサル・キャラクタ名`\unnnn`は、4 桁の短識別子が `nnnn` (8 桁の短識別子が `0000nnnn`) である文字を示します。

ユニバーサル・キャラクタ名では、短識別子が 00A0 未満で、0024 (\$)、0040 (@)、または 0060 (')以外の文字や、D800 ~ DFFF の文字を指定することはできません。

有効なユニバーサル・キャラクタ名の一覧は、付録 F を参照してください。

1.4 コメント

文字定数、文字列リテラル、またはコメント内を除き、`/*` という文字の組はコメントの開始を示します。また、コメントは `/*` という文字の組で終了します。このようなコメントの内容は、マルチバイト文字を識別したり、コメントの終了を示す`*/`を見つける目的でのみ調べられます。

C 言語の要素

1.4 コメント

または、`//`という文字の組は、コメントの始まりを示します。このコメントには、次の改行文字までのすべてのマルチバイト文字 (その改行文字を除く) が含まれます。このようなコメントの内容は、マルチバイト文字を識別したり、終了の改行文字を見つける目的でのみ調べられます。

コメントはネストできません。一度コメントを開始すると、コンパイラは最初の `*/` をコメントの終わりとして処理します。

コードのセクションをコメントにする場合、`/*` と `*/` のシーケンスは使用しないでください。`/*` と `*/` のシーケンスは、コメントを含まないコード・セクションにのみ有効です。次の例の場合には、`#if` と `#endif` の前処理命令を使用してください。

```
#if 0
/* This code is excluded from execution because ... */
code_to_be_excluded ();
#endif
```

`#if` および `#endif` の前処理命令についての詳細は、第 8 章を参照してください。

コメントは、2 つのソース・ファイルにまたがって指定することはできません。ただし、1 つのソース・ファイルにおいてはどんな長さであっても構いません。コメントは、コンパイラとプリプロセッサの両方において空白として解釈されます。

例:

```
"a//b"           // four-character string literal
#include "//e"     // undefined behavior
// */           // comment, not syntax error
f = g/**/h;       // equivalent to f = g / h;
//\
i();              // part of a two-line comment
/\
/ j();           // part of a two-line comment
#define glue(x,y) x##y
glue(/,/ ) k();   // syntax error, not comment
/**/ l();        // equivalent to l();
m = n/**/o
+ p;             // equivalent to m = n + p;
```


1.5 キーワード

C 言語にはいくつかのキーワードが定義されており、各キーワードはコンパイラに対して特別な意味を持ちます。キーワードは、文構成要素を識別して基本型と記憶域クラスを指定します。キーワードを識別子として使用することはできず、宣言することもできません。

表 1-3 は C 言語のキーワードの一覧です。

表 1-3 キーワード

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while
_Bool	_Complex (<i>Alpha only</i>)	inline	restrict
_Imaginary			

コンパイラは、表 1-3 に示したキーワード以外にも、2 つのアンダスコア (_)、またはアンダスコア 1 文字と大文字 1 文字の順で始まる識別子をすべて予約しています。ユーザ変数名の先頭は、このシーケンスにならないようにしてください。

キーワードは、次の目的のために使用します。

- auto , extern , register , static
記憶域クラスを変数または関数に割り当てる。
- _Bool , char , _Complex (*Alpha only*) , const , double , enum , float , int , long , short , signed , struct , union , unsigned , void , volatile
型を構成するか、または修飾する。

C 言語の要素

1.5 キーワード

- `break`, `case`, `continue`, `default`, `do`, `else`, `for`, `goto`, `if`, `return`, `switch`, `while`

文の一部として使用する。

- `typedef`

新しく命名した型を定義する。

- `sizeof`, `__typeof__`

演算を行う。

次の *VAX C* キーワードも、コンパイラに認識されること¹があります。

```
_align
globaldef
globalref
globalvalue
noshare
readonly
variant_struct
variant_union
```

次の *C99* 規格のキーワードも、コンパイラに認識されること²があります。

```
inline
restrict
```

キーワードを余分なマクロ名として使用しないでください。ただし、基本データ型の省略時の大きさを変更する場合は例外です。次にその例を示します。

```
#define int short
```

`int` キーワードを `short` として再定義します。これにより、`int` データ型で宣言されるすべてのデータ・オブジェクトが `short` オブジェクトとして格納されます。

¹ *OpenVMS* システムでは、コンパイラのコマンド行で `/STANDARD=RELAXED_ANSI` (省略時の値), `/STANDARD=VAXC`, `/ACCEPT=VAXC_KEYWORDS` のいずれかを指定した場合に認識されます。 *Tru64 UNIX* システムでは、コンパイラのコマンド行で `-vaxc` または `-accept vaxc_keywords` を指定した場合に認識されます。

² *OpenVMS* システムでは、コンパイラのコマンド行で `/STANDARD=RELAXED_ANSI` (省略時の値), `/STANDARD=C99`, または `/ACCEPT=C99_KEYWORDS` を指定した場合に認識されます。 *Tru64 UNIX* システムでは、コンパイラのコマンド行で `-std` (省略時の値), `-c99`, または `-accept c99_keywords` を指定した場合に認識されます。

1.6 演算子

演算子は 1 つ、または複数のオペランドでの演算を示すトークンであり、ある結果 (値、指名子、副作用、またはこれらの組み合わせ) を返してきます。演算子は式または定数 (式の 1 形式) です。1 つのオペランドを持つ演算子は単項演算子、2 つのオペランドを持つ演算子は 2 項演算子です。次にその例を示します。

```
x = -b;          /* Unary minus operator */
y = a - c;       /* Binary minus operator */
```

3 つのオペランドを持つ演算子は 3 項演算子と呼びます。

演算子にはすべて優先順位が付けられています。優先順位とは、1 つの文の中でどの演算子を先に評価するかを決定する順位付けです。各演算子の役割および演算子の優先順位の規則については、第 6 章を参照してください。

C 言語の演算子には、複数の文字で構成されるものと 1 文字で構成されるものがあります。C 言語の 1 文字演算子は次のとおりです。

```
! % ^ & * - + = ~ | . < > / ? : , [ ] ( ) #
```

C 言語の複数文字演算子は次のとおりです。

```
++ -- -> << >> <= >= == != *= /=
%= += -= <<= >>= &= ^= |= ## && ||
```

および ## の演算子は前処理マクロ定義でのみ使用できます。あらかじめ定義されたマクロと前処理命令についての詳細は、第 8 章を参照してください。

もう 1 つの sizeof 演算子はデータ型のサイズを決定します。sizeof 演算子についての詳細は、第 6 章を参照してください。

旧形式の複合代入演算子 (+= , -= , *= , /= , %= , <<= , >>= , &= , ^= , および |=) は、ANSI C 規格ではサポートされません。これらの演算子をプログラム中で使用した場合の演算結果は保証できません。次にその例を示します。

```
x = -3;
```

この構文では x に -3 が代入され、x-3 は代入されません。

旧形式の複合代入演算子を見つけると、エラー検査のコンパイル・オプションが警告メッセージを表示します。

1.7 区切り記号

C 言語の文字のいくつかは区切り記号として使用され、構文および意味上で独自の機能を持っています。区切り記号は演算子でも識別子でもありません。表 1-4 は C 言語の区切り記号の一覧です。

表 1-4 区切り記号

区切り記号	用途	例
< >	ヘッダ名	<limits.h>
[]	配列区切り文字	char a[7];
{ }	初期化子並び、関数本体、 または複文区切り文字	char x[4] = { 'H', 'i', '!', '\0' };
()	関数の仮引数並び区切り文 字、または式のグループ分け	int f (x,y)
*	ポインタ宣言	int *x;
,	実引数並びセパレータ	char x[4] = { 'H', 'i', '!', '\0' };
:	文ラベル	labela: if (x == 0) x += 1;
=	宣言初期化子	char x[4] = { "Hi!" };
;	文の終了	x += 1;
...	可変個の実引数並び	int f (int y, ...)
#	前処理命令	#include <limits.h>
' '	文字定数	char x = 'x';
" "	文字列リテラルまたは ヘッダ名	char x[] = "Hi!";

次の区切り記号は、組み合わせて使用する必要があります。

< >
[]
()

```
 / /  
 " "  
 { }
```

文字の中には、区切り記号や演算子または演算子の一部のいずれとしても使用できるものがあります。このような文字は、文脈中の発生箇所によって意味が決まります。区切り記号は通常、表 1-4 に示しているように C 構成要素のそれぞれの型を区切るものです。

1.8 文字列リテラル

文字列とは 0 または 1 文字以上の文字のシーケンスです。文字列リテラルは、二重引用符で囲まれた、0 個以上のマルチバイト文字の列です ("xyz" など)。文字列リテラルには、任意の有効文字 (空白文字と文字エスケープ・シーケンスを含む) を含めることができます。ワイド文字列リテラルは、L という文字が前に付くことを除けば、同じです。いったん文字列に文字列リテラルを格納した後に、その文字列を変更した場合の動作は規定されていません。

次の例において、ABC は文字列リテラルです。これは 1 つの文字配列に割り当てられ、この文字配列には文字列リテラルの各文字が 1 つずつ配列要素として格納されます。文字配列に文字列リテラルを格納することにより、その配列の文字を修正できるようになります。

```
char x[] = "ABC";
```

通常は char (文字列リテラルが L で始まる場合は wchar_t) 型の配列として格納され、静的記憶域存続期間を持ちます。

次の宣言は、文字配列を宣言して文字列 "Hello!" を保持しています。

```
char s[] = "Hello!";
```

s 文字配列は二重引用符内に指定した文字で初期化され、ヌル文字 '\0' で終了します。ヌル文字は各文字列の終わりを示し、コンパイラにより文字列リテラルの終わりに自動的に追加されます。行の終わりに 1 つのヌル文字が追加された隣接の文字

列リテラルは自動的に連結され、行継続文字 (行の終わりのバックスラッシュ) をつける必要はなくなります。

通常の文字列リテラルとワイド文字列リテラルは、連結させることができ、その場合、通常の文字列が拡張されてワイド文字列になり、結果としてワイド文字列が生成されます。

次に、有効な文字列リテラルの例を示します。

```
""          /* Here's a string with only the null character */
"You can have many characters in a string."
"\You can mix characters and escape sequences.\n"
"Long lines of text can be continued on the next line \
by using the backslash character at the end of a line."
"Or, long lines of text can be continued by using "
"ANSI's concatenation of adjacent string literals."
"\'\n"      /* Only escape sequences are in this string */
```

文字列リテラルの長さ (ヌル文字は含まない) を調べるには、`strlen`関数を使用します。文字列処理で使用可能なその他のライブラリ・ルーチンについての詳細は、第9章を参照してください。

1.9 定数

C 言語には次の4種類の定数があります。

- 整数定数 (63, 0, 42L など)
- 浮動小数点定数 (1.2, 0.00, 77E+2 など)
- 16進浮動小数点定数 (1/2 を表す0x1P-1や0x.1P3など)
- 文字定数 ('A', '0', L'\n' など)
- 列挙定数 (enum boolean { NO, YES }; など。この場合は NO と YES が列挙定数。)

上記の4つの定数について、次に説明します。

どの定数の値も、指定する型が表現可能な値の範囲内であることが必要です。定数は、その型の種類に関係なく変更しないリテラル値またはシンボル値です。また、定数は第 2.14 節に定義されているように右辺値です。

1.9.1 整数定数

整数定数はすべての数を表すために使用します。整数定数は 10 進基数、8 進基数、または 16 進基数で指定することができます。またオプションによって、基数を指定する接頭語と型を指定する接尾語を含めることもできます。整数定数には、ピリオドまたは指数部を含めることはできません。

整数定数を指定する場合には、次の規則に従ってください。

- 10 進定数を指定するには、先頭が 0 以外の 10 進数字のシーケンスを使用する。10 進定数の値は基数 10 で計算される。
- 8 進定数を指定するには 0 で開始して、その後に数字の 0 ~ 7 で構成されるシーケンスを続ける。先頭の 0 は、8 進数字の 0 を意味する。8 進定数の値は基数 8 で計算される。
- 16 進整数定数を指定するには 0 で開始して、その後に X (または x) 文字を続ける。X または x の後には、1 つ以上の 16 進文字 (0 ~ 9 の数字と A ~ F の大文字または小文字) を続ける。16 進定数の値は基数 16 で計算される (A ~ F の文字が 10 ~ 15 のそれぞれの値に相当する)。

値に L, l, LL (*Alpha only*), ll (*Alpha only*), U, または u が接尾語として付かない場合は、整数定数の型は特に指定しない限り定数値を保持できる最小の型になります。

C99 規格では、表現するために少なくとも 64 ビットを必要とする値の範囲を持つ標準の整数型として、long long int 型 (符号付きと符号なしの両方) を導入しました。Alpha システム上の *Compaq C* では、long long 型を言語拡張として以前のリリースで実装していましたが、コンパイラは整数定数の型を決定するときには C89 規則に従っていました。この規則では、接尾語なしの 10 進整数に対し、signed long で表すには大きすぎるが unsigned long に収まる値の場合には unsigned long 型が割り当てられ、unsigned long で表すには大きすぎる値の場合

だけlong long型が割り当てられます。(注意: long longデータ型がサポートされるのは、Alpha システムだけです。)

long long型を標準化する際に、C99 規格ではこれらの規則が調整され、さらに長い型に拡張できるようになりました。特に、接尾語なしの 10 進整数定数には、その値を保持できる最小の符号付き整数型が割り当てられます (最小の型は、int のままです)。signed long longの最大値よりも大きい値の場合、次に大きい符号付き整数型が処理系で定義されていれば、この型が割り当てられます。このような型がない場合、C99 では動作を未定義としています。ただし、Compaq C では、unsigned long long型を次に使用します。10 進定数に符号なしの型を指定する方法で移植性があるのは、u か U を含む接尾語を使用する方法だけです。

Compaq C は、VAXC モード、COMMON モード、および Strict ANSI89 モード (MIA を含む) では C89 規則を引き続き使用しますが、その他のすべてのモードでは新しい C99 規則を使用します。表 1-5 に、整数定数の型を決定する規則を示します。整数定数の型は、対応するリスト中で、その値を表現できる最初の型になります。

表 1-5 整数定数に型を割り当てる規則

接尾語	10 進定数	8 進定数または 16 進定数
なし	int	int
	long int	unsigned int
	long long int ¹	long int
	unsigned long long int	unsigned long int
		long long int
u または U		unsigned long long int
	unsigned int	unsigned int
	unsigned long int	unsigned long int

¹VAXC、COMMON、ANSI89、および MIA のモードでは、long long intの前にunsigned long intが割り当てられます。

(次ページに続く)

表 1-5 (続き) 整数定数に型を割り当てる規則

接尾語	10 進定数	8 進定数または 16 進定数
	unsigned long long int	unsigned long long int
l または L	long int	long int
	long long int ¹	unsigned long int
	unsigned long long int	long long int
		unsigned long long int
u または U と	unsigned long int	unsigned long int
l または L の両方	unsigned long long int	unsigned long long int
ll または LL	long long int	long long int
	unsigned long long int	unsigned long long int
u または U と	unsigned long long int	unsigned long long int
ll または LL の両方		
¹ VAXC, COMMON, ANSI89, および MIA のモードでは, long long intの前にunsigned long intが割り当てられます。		

たとえば, 定数 59 には int データ型が割り当てられます。定数 59L には long データ型が割り当てられます。定数 59UL には unsigned long int データ型が割り当てられます。

整数定数値は常に 0 または正の整数です。先頭のマイナスは定数の一部ではなく, 単項演算子として解釈されます。値が最大整数値を超えた場合 (オーバーフローを起こした場合) には, コンパイラは警告メッセージを出し, その整数型の最大表現可能な値を使用します。接尾語の付いていない整数定数は, 明示的な指定なしでその定数を最小の整数型で表せるため, 複数の異なる型が割り当てられることがあります。

注意

整数定数の型を決定する新しい C99 規則を使用すると、以前のバージョンのコンパイラで符号なしの型が割り当てられていた定数が、符号付きの型として解釈されることがあります。この違いにより、プログラムの動作に多少影響がある可能性があります。新しいメッセージ `intconstsigned` を使用可能にして、C99 規則では以前のリリースとは異なった方法で扱われる定数がソース・コード内にあることを報告させることができます。このメッセージは、新しいメッセージ・グループ NEWC99 の一部です。符号なしとして扱われることをプログラムが前提としている場合は、"U" または "u" を含む正しい接尾語を追加し、定数が期待どおりの型になるように強制して簡単に修正できます。このように変更することで、旧バージョンとの互換性を持たせ、移植性を確保できます。

1.9.2 浮動小数点定数

浮動小数点定数には有効数字部があり、この後に指数部と、その型を指定する接尾語 (オプション) を続けることができます (例: `32.45E2`)。

有効数字部の構成要素としては、整数部を表す数字列、その後にピリオド(.)、その後には小数部を表す数字列があります。

指数部の構成要素としては、`e`、`E`、`p`、または `P`、その後に数字列 (オプションで符号付き) からなる指数があります。

有効数字部の整数部と小数部のいずれかが存在しなければなりません。10 進浮動小数点定数の場合、ピリオドと指数部のいずれかが存在しなければなりません。

1.9.2.1 意味

浮動小数点定数の有効数字部は、10 進または 16 進の有理数として解釈されます。指数部の数字列は、10 進整数として解釈されます。10 進浮動小数点定数の場合、指数部は 10 の累乗を示し、これにより有効数字部がスケールリングされます。16 進浮動小数点定数の場合、指数部は 2 の累乗を示し、これにより有効数字部がスケールリングされます。10 進浮動小数点定数、および `FLT_RADIX` が 2 の累乗でない 16 進浮動小数点定数の場合、プラットフォーム依存の選択方法により、結果は表現可能な最も近い値か、表現可能な最も近い値に隣接する表現可能な値 (大きい側の値

または小さい側の値) になります。FLT_RADIX が 2 の累乗の 16 進浮動小数点定数の場合、結果は正確に丸められます。

浮動小数点定数は負でない数でなければなりません。先頭のマイナスは定数部には含められず、単項演算子と解釈されます。

1.9.2.2 浮動小数点型

浮動小数点定数には、次の型があります。

- 接尾語なしの浮動小数点定数の場合、double型。
値がdouble型で表現できる最大値を超えた場合、コンパイラはオーバーフローの警告を出力します。(結果は、double型に納まるように切り詰められます。)
- 接尾語が文字 f または F の場合、float型。
- 接尾語が文字 l または L の場合、long double型。

1.9.2.3 16 進浮動小数点定数

16 進浮動小数点定数について簡単に説明します。C99 規格で、浮動小数点定数の 16 進形式が導入されました。この形式の定数によって、浮動小数点値が、仮数部の最下位ビットまで正確に記述できるようになりました。この形式では、表現にビット・パターンは指定しません。代わりに、通常の 10 進浮動小数点定数と同じように解釈されます。ただし、有効数字部は 16 進数で書き、指数部は有効数字部を乗算する 2 の累乗を示す 10 進整数として表現されます。有効数字部と指数部は、"E"ではなく"P"で区切ります。このため、たとえば 1/2 は、0x1P-1 または 0x.1P3 と書くことができます。

C99 規格では、この形式の値用に、printf/scanfの指定子も追加しています。ただし、この機能は、OpenVMS Version 7.3 以降の OpenVMS 実行時ライブラリでサポートされます。

1.9.2.4 例

表 1-6 に、有効な表記オプションの例を示します。

表 1-6 浮動小数点の表記

表 記	値	型
.0	0.000000	double
0.	0.000000	double
2.	2.000000	double
2.5	2.500000	double
2e1	20.00000	double
2E1	20.00000	double
0x1P-1	0.500000	double
0x.1P3	0.500000	double
2.E+1	20.00000	double
2e+1	20.00000	double
2e-1	0.200000	double
2.5e4	25000.00	double
2.5E+4	25000.00	double
2.5F	2.500000	float
2.5L	2.500000	long double

1.9.3 文字定数

文字定数は、ソース文字集合の任意の文字を一重引用符で囲んだものです。文字定数は `int` 型のオブジェクトで表されます。次にその例を示します。

```
char alpha = 'A';
```

第 1.9.3.3 項で説明するように改行文字、一重引用符、二重引用符、およびバックスラッシュなどの文字は、エスケープ・シーケンスを使用して文字定数に含めることができます。また、第 1.9.3.4 項で説明するように、すべての有効文字を数値エスケープ・シーケンスを使用して定数に含めることもできます。

単一文字を含む文字定数の値は、現在の文字集合におけるその文字の数値です。次に説明するように、一重引用符内に複数の文字を含む文字定数はコンパイラが決定する値を持ちます。8 進または 16 進のエスケープ・シーケンスで表される文字定数値は、エスケープ・シーケンスの 8 進値または 16 進値と同じです。ワイド文字定数の値は、`mbtowc` ライブラリ関数が決定します (第 1.9.3.1 項を参照してください)。

1 つの文字定数は、最大 4 文字という制限があります。一重引用符で 4 文字以上の文字を囲むと (例: `'ABCDE'`)、オーバーフロー警告が出ます。

文字定数の順番を決定するバイトはプラットフォームにより異なりますので、ご注意ください。

1.9.3.1 ワイド文字

C 言語はワイド文字を使用して拡張文字集合を提供します。ワイド文字とは、`char` 型には格納できない大きい文字のことです。`wchar_t` 型は通常、257 文字以上の可能な文字を必要とする文字集合の文字定数を表すために使用します。この理由は、8 ビットでは最大で 256 までの異なる値しか表すことができないからです。

拡張文字集合の文字定数は、先頭に `L` を付けて記述し、ワイド文字定数と呼ばれます。ワイド文字定数は `wchar_t` 整数型であり、`<stddef.h>` ヘッダ・ファイルに定義されています。ワイド文字定数は、通常のエスケープ・シーケンスのように 8 進文字または 16 進文字のエスケープ・シーケンスで表すことができますが、先頭に `L` を付ける必要があります。

ワイド文字で構成される文字列を形成することもできます。コンパイラは文字列が `wchar_t` 型の配列であるかのように記憶域を割り当て、ワイド・ヌル文字 `'\0'` を文字列の終わりに付けます。この配列は文字列の文字およびワイド・ヌル文字を格納できる長さであり、指定した文字で初期化されます。

次の例は、有効なワイド文字定数と文字列リテラルを示しています。

```
wchar_t wc = L'A';  
wchar_t wmc = L'ABCD';  
wchar_t *wstring = L"Hello!";  
wchar_t *x = L"Wide";  
wchar_t z[] = L"wide string";
```

*Compaq C*コンパイラは、32 ビットの記憶域に unsigned long オブジェクト (*OpenVMS*) または unsigned int オブジェクト (*Tru64 UNIX*) として wchar_t オブジェクトを格納します。ワイド文字の終わりのヌル文字は 32 ビットの長さです。

1.9.3.2 多バイト文字

非 ASCII 文字を通常の 8 ビットの char サイズで表すために、シフト依存のコード化された方式を使用した拡張文字集合を使用することができます。このコード化によって、多バイト文字が生成されます。ANSI C 規格では、wchar_t ワイド文字型を提供するとともに、このコード化された方式をサポートしています。

Compaq C では ANSI 規格に準拠して、次の文脈で多バイト文字を認識します。

- コメント
- 文字列リテラル
- ヘッダ名
- 文字定数

多バイト文字のコードの入出力を適切に行ったり、既存の文字列処理ルーチンとの矛盾を回避するには、多バイト文字を使用する際に次の規則に従う必要があります。

- すべてのビットを 0 にセットしたバイトは、常にヌル文字として認識される。ヌル文字は単一バイトにのみ使用可能。
- ヌル文字は、多バイト文字の 2 番目以降のバイトとして使用することはできない。

多バイト文字をワイド文字定数とワイド文字列リテラルに変換すると、シフト状態のコード化を処理する際に、プログラミング上の問題が少なくなります。多バイト文字をワイド文字に変換するために、いくつかの C ライブラリ関数を使用できます。詳細については、第 9 章を参照してください。

1.9.3.3 文字エスケープ・シーケンス

標準端末に表示できない文字，または文字定数や文字列リテラルに使用されている場合に特定の意味を持つ文字は，文字エスケープ・シーケンスとして入力することによって，ソース文字として入力できます。バックスラッシュ (\) は，各文字エスケープ・シーケンスを開始します。各エスケープ・シーケンスは，char または wchar_t オブジェクトに格納されます。表 1-7 は，ANSI 定義のエスケープ・シーケンスの一覧です。

表 1-7 文字エスケープ・シーケンス

文字	エスケープ・シーケンス
警告 (ベル)	\a
バックスペース	\b
フォーム・フィード	\f
改行	\n
キャリッジ・リターン	\r
水平タブ	\t
垂直タブ	\v
バックスラッシュ	\\
一重引用符	\'
二重引用符	\"
疑問符	\?

これ以外の文字エスケープ・シーケンスは無効です。ソース・コード内にこれ以外のシーケンスを見つけると，コンパイラは警告を出し，バックスラッシュ文字は無視されます。

次に，文字エスケープ・シーケンスを使用した例を示します。

```
printf ("\t\aReady\?\n");
```

これを実行すると警告ベルが鳴り，次のプロンプトが表示されます。

```
Ready?
```

1.9.3.4 数値エスケープ・シーケンス

コンパイラはすべての文字を整数表現として扱うため、ソース・コードの任意の文字をそれに相当する数値で表現することができます。これを、数値エスケープ・シーケンスと呼びます。この文字は、バックスラッシュ (\) を入力し、それに続けて現在の文字集合に相当する 8 進または 16 進の整数を記述します (ASCII 等価テーブルについては、付録 C を参照してください)。たとえば、ASCII 文字集合を使用する場合、文字の A は \101 (8 進表現) または \x41 (16 進表現) と表現することができます。数値エスケープ・シーケンスでは 8 進の値が省略時の値ですから、この例では前に 0 を付ける必要はありません。バックスラッシュの後の小文字の x は 16 進表現を示します。たとえば、\x5A は文字 Z と同値です。

次に、数値エスケープ・シーケンスの例を示します。

```
#define NULL '\0' /* Defines logical null character */
char x[] = { '\110', '\145', '\154', '\154', '\157', '\41', '\0' };
/* Initializes x with "Hello!" */
```

エスケープ・シーケンスは、8 進数の場合は 3 つの数字を展開するか、または最初の文字以外の 8 進数以外文字が見つかるまで展開します。たとえば、文字列 "\089" は \0, 8, 9, および \0 の 4 文字と解釈されます。

16 進エスケープ・シーケンスの場合、エスケープ・シーケンスの文字数に制限はありません。ただし、16 進の値が通常の文字定数の unsigned char 型で表すことのできる最大値、またはワイド文字定数の wchar_t 型で表すことのできる最大値を超える場合の結果は保証できません。たとえば、'\x777' は不正な例です。

また、3 文字を超える文字数を持つ 16 進エスケープ・シーケンスは、エラー検査のコンパイル・オプションを使用すると警告が出されます。

16 進エスケープ・シーケンスの後に 16 進数を指定するために、文字列の連結を使用することができます。次の例では、a は両方の場合で同じ値に初期化されます。

```
char a[] = "\xff" "f";
char a[] = { '\xff', 'f', '\0' };
```


実行するマシンが異なる文字集合を使用している場合、数値エスケープ・シーケンスを使用して作成されたプログラムは移植できません。算術演算を整数文字値で実行すると、移植時にまた別の問題が発生します。これは、マシンごとに複数文字定数 (例: 'ABC') の表現が異なるからです。

1.9.4 列挙定数

列挙型は、1 つ以上の列挙定数を指定して許容値を定義します。列挙定数は `int` 型を持ちます。列挙型の宣言と使用についての詳細は、第 3.6 節を参照してください。

1.10 ヘッダ・ファイル

ヘッダ・ファイルとは、コンパイル時にソース・ファイルに取り込まれるテキスト・ファイルです。コンパイル時にヘッダ・ファイルを取り込むには、`#include` 前処理命令をソース・ファイルに使用します。命令についての詳細は、第 8 章を参照してください。`#include` 前処理命令は、内容とは関係なくヘッダ・ファイル全体に置き換わります。

別ファイルを取り込むために、ヘッダ・ファイルに他の `#include` 前処理命令を含めることができます。`#include` 命令は任意の深さでネストできます。

ヘッダ・ファイルには任意の有効な C ソース・コードを取り込むことができます。ヘッダ・ファイルは、外部変数宣言、マクロ定義、型定義、および関数宣言を取り込むために使用します。`stdio.h` ヘッダ・ファイルに C ライブラリ入出力関数が宣言されているように、論理的に関連する関数のグループは 1 つのヘッダ・ファイル内に一緒に宣言されます。ヘッダ・ファイルには、習慣として `.h` という接尾語 (例: `stdio.h`) を付けます。

ヘッダ・ファイルにこれらの区切り文字を使用できることが定義されていないので、ヘッダ・ファイル名として `'`、`\`、`"`、または `*` の文字を使用することはできません。

プログラムで参照する場合には、次の例に示すようにヘッド・ファイル名は山括弧または二重引用符で囲みます。

```
#include <math.h> /* or */  
#include "local.h"
```

2つの書式の相違については、第8章で説明します。指定されたファイルを検索するためにコンパイラが使用するアルゴリズムについては、第B.37節で説明します。ANSI規格ヘッド・ファイルの各ライブラリ・ルーチンについては、第9章で説明します。

1.11 制限事項

ANSI C規格では、C言語の使用について環境上の制限をいくつか挙げています。この制限は、Cコンパイラの規格合致の処理系のために最小限の規格を定義しようとしたものです。たとえば、識別子における有効文字数は、ANSI C規格が必要とする最小限のセットにより処理系が定義するものです。

また、ANSI C規格には、汎整数型と浮動小数点型の特性を制限する数値制限も含まれています。ほとんどの場合、このような制限はC言語またはコンパイラの使用には影響を及ぼしません。しかし、通常とは異なる大きいプログラムまたは通常とは異なる構成のプログラムの場合に、特定の制限に到達する可能性があります。ANSI規格には最小限の制限があるので、*Compaq C*での実際の制限についてはプラットフォームに固有の*Compaq C*マニュアルをご覧ください。

1.11.1 翻訳上の制限

ANSI C規格で示されているように、*Compaq C*の処理系ではすでに翻訳上の多くの制限が取り除かれ、アプリケーションをより自由に使用することができます。*Compaq C*の制限は次のとおりです。

- 内部識別子またはマクロ名の有効文字数は、最大で32,767文字である。これを超えると、警告メッセージが出される。
- *Tru64 UNIX*システムの場合、外部識別子の有効文字数は最大で1023文字である。

- *OpenVMS VAX*プラットフォームの場合、外部識別子の有効文字数は最大で 31 文字である。これを超えてその識別子が切り捨てられた場合、警告メッセージが出される。
- *OpenVMS*システムでは関数の実引数/仮引数は最大で 253 個であり、*Tru64 UNIX*システムでは関数の実引数/仮引数は最大で 1023 個である。
- 1 つの関数の実引数は最大で 1012 バイトである。*OpenVMS*システムの場合の関数の実引数並びは、最大で 1012 バイトである。
- 論理ソース行は最大で 32,767 文字である。
- 物理ソース行は最大で 32,767 文字である。
- 文字列リテラルの表現は、最大で 32,767 バイトである。ただし、この制限は連結の結果として形成される文字列リテラルには適用されない。

1.11.2 数値制限

数値制限は、整数型と浮動小数点型のサイズと特性を定義します。数値制限については、`limits.h` および `float.h` ヘッド・ファイルに記述されています。制限は次のとおりです。

- `char`型の各文字は 8 ビットで表現される。
- `wchar_t`型の各文字は 32 ビットで表現される。
- `char`型のマシン表現と有効値は `signed char`型の場合と同様。コンパイル・コマンド行オプションは、この同値を `unsigned char` に変更する。
- *OpenVMS*システムでは、`int`型および `signed int`型のマシン表現と有効値は `long int`型の場合と同様。
- *OpenVMS*システムでは、`unsigned int` のマシン表現と有効値は `unsigned long int`型の場合と同様。
- *Tru64 UNIX*システムでは、`long int`型と`unsigned long int`型は 64 ビットであり、`int`型と`unsigned int`型は 32 ビットである。
- `long double`型のマシン表現と有効値は `double`型の場合と同様。

1.11.3 文字表示

実行文字集合の文字は、画面またはファイルの現在印字位置に出力されます。現在印字位置は、次の出力文字が現れる場所として ANSI C 規格で定義されています。文字が出力されると、現在印字位置が現在行の次の位置 (左または右) に進みます。

*Compaq C*コンパイラは、出力行を超えて左から右へ現在印字位置を移動させます。

C 言語は、オペレーティング・システムの記述に使用するために、最初は小規模で移植可能なプログラミング言語として設計されました。その後現在に至るまでに、あらゆる種類のプログラムを記述するための強力なツールに成長を遂げました。C 言語は、最良のプログラミングを行うために次の機能を提供します。

- 単語要素の基本的なセット
- 次に示すような広範囲にわたるデータ・オブジェクトの型
 - 整数および浮動小数点の定数と変数
 - メモリ内のデータ位置を示すポインタ，およびポインタ算術機能
 - 同一の型を持つデータの配列
 - 異なるデータ型のメンバを持つ構造体と共用体
- 独立した個々のコード・ブロックを名前付きの関数へグループ分けする機能
- ビット演算子を含む式を形成するために使用する演算子の大規模なセット
- データ・オブジェクトと関数を宣言するための簡単な方法
- 言語の機能性を拡張するための複数の前処理命令
- 多くの共通プログラミング・タスクを処理するさまざまなライブラリ関数
- 高度な移植性

この章では、C 言語の機能を十分に活用するための C 言語の基本事項について説明します。

- ブロック (第 2.1 節)
- コンパイル単位 (第 2.2 節)
- スコープ (第 2.3 節)

基本事項

- 可視性 (第 2.4 節)
- 副作用と評価順序点 (第 2.5 節)
- 不完全型 (第 2.6 節)
- 互換型と合成型 (第 2.7 節)
- 結合 (第 2.8 節)
- 記憶域クラス (第 2.10 節)
- 記憶域クラス修飾子 (第 2.11 節)
- 順方向参照 (第 2.12 節)
- タグ (第 2.13 節)
- 左辺値と右辺値 (第 2.14 節)
- 名前空間 (第 2.15 節)
- 前処理 (第 2.16 節)
- 型名 (第 2.17 節)

これらの各節で、C 言語に出てくる用語と基本事項について説明します。この基本事項を理解することによって、C 言語を使用して作業を進めるための適切な基礎知識を得ることができ、基本事項と C 言語中のより複雑な事項との関係を理解する上で役に立ちます。

2.1 ブロック

C 言語のブロックとは、中括弧 (`{}`) で囲まれたコード・セクションを表します。ブロックの定義を理解することは、スコープ、可視性、外部宣言、および内部宣言といった C 言語に出てくる他の多くの事項を理解するために非常に重要です。

次に、2 つのブロックの例を示します。この例では、あるブロックが別のブロック中で定義されています。

```
main ()
{
    /* This brace marks the beginning of the first block */
    int x;
    if (x!=0)
    {
        /* This brace marks the beginning of the second block */
        x = x++;
        return x;
    };
    /* This brace marks the end of the inner block */
}
/* This brace marks the end of the outer block */
```

また、ブロックは複文でもあり、関連した一連の C 言語の文を中括弧で囲んだ形で記述します。プログラムに使用するオブジェクトの宣言は 1 つのブロック内の任意の位置に指定でき、オブジェクトのスコープと可視性に影響を及ぼす可能性があります。スコープについては第 2.3 節を、可視性については第 2.4 節を参照してください。

2.2 コンパイル単位

コンパイル単位は、コンパイルされ、1 つの論理単位として扱われる C ソース・コードです。コンパイル単位は通常、1 つまたは複数のファイルです。しかし、ファイルの一部を選択してそれをコンパイル単位にすることもできます。たとえば、`#ifdef` 前処理命令を使用して、特定のコード・セクションを選択することができます。コンパイル単位内の宣言と定義は、関数とデータ・オブジェクトのスコープを決定します。

`#include` 前処理命令を使用して取り込んだファイルは、コンパイル単位の一部となります。条件付きの取込み前処理命令によって省かれたソース行は、コンパイル単位には含まれません。

コンパイル単位は、識別子のスコープを決定し、他の内部識別子および外部識別子への識別子の結合を決定する上で重要な要素です。スコープについては第 2.3 節を、結合については第 2.8 節を参照してください。

コンパイル単位は、他のコンパイル単位のデータまたは関数を次の方法で参照することができます。

- あるコンパイル単位の関数で異なるコンパイル単位の関数を呼び出す。

- データ・オブジェクトに外部結合を割り当てて、他のコンパイル単位がそのデータ・オブジェクトにアクセスできるようにする。第 2.8 節を参照のこと。

複数のコンパイル単位からなるプログラムをそれぞれ別々にコンパイルし、後でリンクして、実行可能なプログラムを作成することができます。有効な C コンパイル単位は、最低 1 つの外部宣言からなります。第 4.3 節を参照してください。

厳密な ANSI 規格モードを除くモードでは、宣言を持たない翻訳単位はコンパイラから警告が出されますが、受け付けられます。

2.3 スコープ

識別子のスコープとはプログラムの範囲であり、宣言された識別子はその範囲内で意味を持ちます。識別子は、コンパイラによって認識された場合に意味を持ちます。スコープは、識別子の宣言の位置によって決定されます。スコープ外の識別子にアクセスしようとすると、エラーが生じます。各宣言は、次の 4 種類のスコープのいずれかを持っています。

- ファイル
- ブロック
- 関数
- 関数プロトタイプ (関数の仮引数型のみを含む宣言)

列挙定数のスコープは、列挙子並びにある列挙子の定義で始まります。文ラベルのスコープには、関数本体全体が含まれます。他の型の識別子のスコープは、その識別子の宣言にある識別子自体で始まります。識別子のスコープを理解した上で、これ以降の各項を参照してください。

2.3.1 ファイル・スコープ

宣言の位置が任意のブロックまたは関数の仮引数並びの外側にある識別子は、ファイル・スコープを持っています。ファイル・スコープを持つ識別子は、内部ブロック宣言によって隠されていない限り、識別子の宣言からコンパイル単位の終了まで

の間で参照できます。次の例では、off識別子はファイル・スコープを持っています。

```
int off = 5;      /* Declares (and defines) the integer
                  identifier off.                      */
main ()
{
    int on;       /* Declares the integer identifier on.      */
    on = off + 1; /* Uses off, declared outside the function
                  block of main. This point of the
                  program is still within the
                  active scope of off.                      */

    if (on<=100)
    {
        int off = 0; /* This declaration of off creates a new object
                      that hides the former object of the same name.
                      The scope of the new off lasts through the
                      end of the if block.                      */

        off = off + on;
        return off;
    }
}
```

2.3.2 ブロック・スコープ

ブロック内または関数定義の仮引数並び内に現れる識別子は、ブロック・スコープを持ち、内部ブロック宣言によって隠されていない限りブロック内で参照できます。

ブロック・スコープは識別子の宣言で始まり、右中括弧 (}) で終了します。次の例では、red識別子はブロック・スコープを持ち、blue はファイル・スコープを持っています。

```
int blue = 5;      /* blue: file scope                      */
main ()
{
    int x = 0 , y = 0; /* x and y: block scope          */
    int red = 10;      /* red: block scope              */
    x = red + blue;
}
```

2.3.3 関数スコープ

関数スコープを持つのは文ラベルのみです(第7章を参照してください)。関数スコープを持つ識別子は、その識別子が宣言されている関数の範囲内では固有です。ラベル付きの文は goto文の対象として使用され、構文(コロン(:)と文が続くラベル)によって暗黙に宣言されます。次にその例を示します。

```
int func1(int x, int y, int z)
{
    label: x += (y + z); /* label has function scope */
    if (x > 1) goto label;
}
int func2(int a, int b, int c)
{
    if (a > 1) goto label; /* illegal jump to undefined label */
}
```

文ラベルについては、第7.1節を参照してください。

2.3.4 関数プロトタイプ・スコープ

仮引数宣言の関数プロトタイプ並び内に現れる識別子は、関数プロトタイプ・スコープを持っています。この識別子のスコープは識別子の宣言で始まり、関数プロトタイプ宣言並びの終わりで終了します。次にその例を示します。

```
int students ( int david, int susan, int mary, int john );
```

この例では、識別子(david, susan, mary, および john)は宣言で始まり右括弧で終了するスコープを持っています。students関数の型は、「4つのint仮引数を持つintを返す関数」です。これらの識別子は実際、関数が定義された後に使用される実際の仮引数名用のプレースホルダです。

2.4 可視性

識別子は、プログラムの一定の範囲内でのみ可視性があります。識別子はネストされたブロック内で再度宣言することによって上書きされるか、または隠されることがない限りスコープ全体で可視性があります。可視状態の場合にのみ識別子を使用できるため、可視性はデータ・オブジェクトまたは他の識別子へのアクセス機能に影響を与えます。

識別子を特定の目的のために使用した後は、その識別子を同じスコープ内で別の目的のために再び使用することはできません。ただし、異なる名前空間で使用することはできます。名前空間の制限については、第 2.15 節を参照してください。たとえば、ある識別子と同じ名前の異なる 2 つのデータ・オブジェクトを同一のスコープ内で宣言することはできません。

同じ名前を持つ 2 つの識別子のうちの一方のスコープが、もう一方に含まれている (ネストされている) 場合には、内側のスコープを持つ識別子は可視状態のままです。これに対して、より広いスコープを持つ識別子は、識別子の内側のスコープが継続する間は隠された状態になります。

次の例では、`number` 識別子が 2 回使用されています。1 回は整数変数として、もう 1 回は浮動小数点変数として使用されています。`main` 関数内では、`number` 整数は `number` 浮動小数点によって隠されます。

```
#include <math.h>
int number;          /* number is declared as an integer variable */

main ()
{
    float x;
    float number;     /* This declaration of number occurs in an inner
                        block, and "hides" the outer declaration.
                        The inner declaration creates a new object */
    x = sqrt (number); /* x receives a floating-point value      */
}
```

2.5 副作用と評価順序点

Compaq C で有効な演算子のほとんどでは、式の評価順序は指定されていません。評価順序は使用状況に応じてコンパイラ内で決定されるため、ある演算子を使用すると予測できない結果が生じる可能性があります。この予測できない結果は、副作用によって引き起こされます。

オペランドの記憶域に影響を与える演算子には、副作用があります。プログラマは、必要な結果を得るために故意に副作用を引き起こすことができます。実際に、代入演算子はジョブを行うための記憶域を変更したために発生した副作用です。Compaq C では、指定した式のすべての副作用がプログラム内の次の評価順序点までに完了することが保証されています。評価順序点 (シーケンス・ポイント) とはプログラム中にあるチェックポイントであり、ここで式の中の演算が終了しているかどうかをコンパイラが検査します。

最も重要な評価順序点は文の終了を示すセミコロンです。セミコロンに到達すると、すべての式とその副作用は完全に評価されます。その他の評価順序点を次に示します。

- `expr1, expr2` (コンマ演算子)
- `expr1 && expr2` (論理積演算子)
- `expr1 || expr2` (論理和演算子)
- `expr1 ? expr2 : expr3` (条件演算子)

上記の演算は評価順序、つまりシーケンスを保証します (この場合、`expr1`、`expr2`、および `expr3` が式です)。各演算子について、式 `expr1` の評価は式 `expr2` (条件式の場合には `expr3`) を評価する前に行われることが保証されます。

式の評価順序がわからない場合は、実行順序に依存する演算は行わないでください。これは、結果に一貫性と移植性がなくなるからです。ある 1 つのデータ・オブジェクトを同じ式の中で複数回使用する場合はもちろんですが、1 回しか使用しない場合でも副作用は発生します。たとえば、次のコードでは代入演算子のオペランドを評価する順序を定義していないため一貫性のない結果となります。

```
int x[4] = { 0, 0, 0, 0 };  
int i = 1;  
x[i] = i++;
```

添字が評価される前に i の増分が行われると, $x[2]$ の値が 1 になります。添字が最初に評価されると, $x[1]$ の値が 1 になります。

関数呼出しにも副作用があります。次の例では, $f1(y)$ と $f2(z)$ の呼び出す順序を定義していません。

```
int y = 0;  
int z = 0;  
int x = 0;  
  
int f1(int s)  
{  
    printf ("Now in f1\n");  
    y += 7;      /* Storage of y affected */  
    return y;  
}  
  
int f2(int t)  
{  
    printf ("Now in f2\n");  
    z += 3;      /* Storage of z affected */  
    return z;  
}  
  
main ()  
{  
    x = f1(y) + f2(z);    /* Undefined calling order */  
}
```

x の値は常に 10 になりますが, `printf`関数はどんな順序でも実行できます。

2.6 不完全型

識別子は, 最初に不完全型として宣言することができます。不完全型宣言はオブジェクトについて宣言しますが, そのサイズを決定するのに必要な情報は提供しません。したがって, サイズのわからない配列の宣言は不完全型宣言です。

```
extern int x[];
```

不完全型は、次の宣言により完全にすることができます。不完全型は主に、順方向参照の配列、構造体、および共用体に使われます (順方向参照については、第 2.12 節を参照してください)。集成型のオブジェクトに不完全型のメンバを入れることはできません。したがって、集成体オブジェクト (構造体または配列メンバ) にそれ自体を入れることはできません。これは、集成型はその宣言が終わるまで完了しないからです。次の例では、どのように不完全構造体型が宣言され、そして完了するかを示しています。

```
struct s
{ struct t *pt }; /* Incomplete structure declaration */
.
.
.
struct t
{ int a;
  float *ps }; /* Completion of structure t */
```

void型は不完全型の特殊な型です。これは完了できない不完全型であり、関数が値を返さないことを表すために使用します。void型については、第 3.5 節を参照してください。

2.7 互換型と合成型

2つの型の互換性は、それぞれの型の類似点に依存します。型の互換性は、型の変換および演算を行う際に特に認識すべき重要なポイントです。同じオブジェクトまたは関数を参照する同じスコープにある有効な宣言は、すべて互換型を持たなければなりません。これら2つの型が次のカテゴリのいずれかにあてはまる場合、それらには互換性があります。

- 2つの型が同じである場合。
- 2つの修飾型 (第 3.7 節を参照のこと) が同様に修飾されており、修飾される前のその2つの型に互換性がある場合。型宣言の修飾子の順序には関係ない。
- short, signed short, short int, および signed short int型は同じ。
- unsigned short および unsigned short int型は同じ。

- `int` , `signed` , および `signed int` 型は同じ。
- `unsigned` および `unsigned int` 型は同じ。
- `long` , `signed long` , `long int` , および `signed long int` 型は同じ。
- `unsigned long` および `unsigned long int` 型は同じ。
- 2つの配列型が同じサイズで互換型の要素を持っている場合。1つの配列のサイズが不明の場合には、互換性のある要素型を持つ他のすべての配列型と互換性がある。
- 2つの共用体または構造体が異なるコンパイル単位で宣言されている場合。それらは同じメンバを同じ順序で持ち、さらにそのメンバが(ビット・フィールドを含む) 同じ幅を持つ。
- 2つの列挙型のすべてのメンバが同じ値を持つ場合。すべての列挙型は他の列挙型と互換性がある。列挙型は `signed int` と互換性がある。
- 2つのポインタ型が同様に修飾され、互換型のオブジェクトを示す場合。
- 旧形式の宣言 (`int tree()` など) を使用して宣言された関数型は、その返却値の型に互換性がある場合には別の関数型と互換性がある。
- 新しいプロトタイプ形式の宣言 (`int tree (int x)` など) を使用して宣言された関数型は、次に示す場合には関数プロトタイプで宣言される別の関数型と互換性がある。
 - 返却値の型に互換性がある。
 - 仮引数の数 (省略記号が使用されている場合は省略記号も含む) が一致している。
 - 仮引数型に互換性がある。修飾型で宣言される各仮引数の場合には、互換性の比較用の型は宣言型のうちの修飾されていないものである。
- 返却値の型に互換性があり、旧形式の関数宣言が定義ではない場合には、プロトタイプ形式の関数宣言の関数型は旧形式の関数宣言の関数型と互換性がある (関数宣言の異なる形式については第5章を参照のこと)。また、次の条件をすべて満たす場合には、プロトタイプ形式の関数宣言の関数型は、旧形式の関数定義の関数型と互換性がある。
 - 2つの関数の返却値の型に互換性がある。

基本事項

2.7 互換型と合成型

- － 仮引数の数が一致する。
- － プロトタイプ形式の関数宣言に，仮引数として省略記号が含まれていない。
- － 旧形式の仮引数の拡張型が，プロトタイプ形式の仮引数型と互換性がある。
次の例では，tree関数と tree2関数には互換性があるが，tree関数と tree1関数，および tree1関数と tree2関数には互換性はない。

```
int tree (int);
int tree1 (char);
int tree2 (x)
    char x; /* char promotes to int in old-style
              function parameters, and so is
              compatible with tree          */
{
    ...
};
```

次の型は，互換性があるように見えますが互換性はありません。

- unsigned int および int型
- char , signed char , および unsigned char型

合成型

合成型は2つの互換型から構成され，2つの型の両方に互換性があります。合成型は次の条件を満たします。

- 1つの型の配列のサイズが明確になっている場合，合成型の配列も同じサイズを持つ。サイズが明確になっている配列がなく，1つの型が可変長配列の場合，合成型もその配列と同じ型である。
- 1つの型のみがプロトタイプを持つ関数型である場合，合成型は仮引数型並びを持つ関数型である。
- 両方の型がプロトタイプを持つ関数型である場合，合成仮引数型並びの各仮引数の型は対応する仮引数の合成型である。

次のファイル・スコープの宣言を例にとります。

```
int f(int (*) (), double (*) [3]);
int f(int (*) (char *), double (*) []);
```


この関数の合成型は次の結果になります。

```
int f(int (*) (char *), double (*)[3]);
```

前述の合成型規則は、合成型から派生した型に再帰的に適用されます。

2.8 結合

データ・オブジェクトおよび関数には、暗黙または明示的に結合を割り当てることができます。結合には次の3種類があります。

- 内部結合

同じコンパイル単位で宣言され、そのコンパイル単位外では認識されないデータ・オブジェクトまたは関数を参照する宣言。

- 外部結合

コンパイル単位外で認識されているデータ・オブジェクトまたは関数の定義を参照する宣言。そのオブジェクトの定義も外部結合を持つ。

- 結合なし

固有のデータ・オブジェクトを宣言する宣言。

同じオブジェクトまたは関数の宣言が複数作成されると、結合が作成されます。結合した宣言は、同じスコープにも異なるスコープにも入れることができます。外部結合されたオブジェクトは、実行可能なファイルの作成に使用される任意のコンパイル単位の任意の関数で使用可能です。内部結合されたオブジェクトは、宣言が現れるコンパイル単位でのみ使用できます。

`static` および `extern` キーワードと結合の概念には関連がありますが、直接の関連はありません。オブジェクトの宣言に `extern` キーワードを使用しても、外部結合は保証されません。次の規則は、オブジェクトまたは関数の実際の結合を決定します。

- `auto` または `register` 記憶域クラスで明示的に指定された識別子は結合を持たない。

- ブロック・スコープと extern 記憶域クラス指定を持つ識別子は、ファイル・スコープを持つ同じ識別子の任意の可視宣言と結合を持つ。オブジェクトまたは関数のこのような宣言が可視状態ではない場合、そのオブジェクトまたは関数は外部結合を持つ。
- 関数の宣言は特に指定しない限りは外部結合を持つ。関数に対して指定可能な唯一の他の記憶域クラスは static である。static は明示的に指定しなければならず、ブロック・スコープの関数宣言には適用できない。このため、これは内部結合になる。
- 明示的な記憶域クラス指定を持たない、または extern 記憶域クラス指定を持つデータ・オブジェクトのファイル・スコープ宣言は外部結合を持つ。
- ファイル・スコープと static 記憶域クラスを持つ識別子は、内部結合を持つ。
- ブロック・スコープを持ち、extern 記憶域クラス指定を持たない識別子は結合を持たない。

データ・オブジェクトと関数以外の識別子は結合を持ちません。また、関数の仮引数として宣言される識別子も結合を持ちません。

次の例は、異なる結合を持つ宣言を示しています。

```
extern int x;           /* External linkage          */
static int y;          /* Internal linkage          */
register int z;         /* Illegal storage-class declaration */
main ()                /* Functions default to external linkage */
{
    int w;             /* No linkage                */
    extern int x;       /* External linkage          */
    extern int y;       /* Internal linkage          */
    static int a;       /* No linkage                */
}
void func1 (int arg1) /* arg1 has no linkage      */
{ }
```

*Compaq C*では同じオブジェクトが内部結合と外部結合の両方で宣言されると、メッセージが出されます。

2.9 仮定義

ファイル・スコープを持つ識別子，初期化子を持たない識別子，記憶域クラス指定子，または static 記憶域クラス指定子を持たない識別子の宣言は仮定義と呼ばれます。オブジェクトの他の定義がコンパイル単位に現れない場合には，仮定義のみが適用されます。この場合，オブジェクトのすべての仮定義はオブジェクトにファイル・スコープ定義が 1 つだけ存在する場合と同様に扱われ，0 で初期化されます。

仮定義されたオブジェクトの定義を後でコンパイル単位に使用すると，その仮定義はオブジェクトの冗長な宣言として扱われます。オブジェクトの識別子の宣言が仮定義であり，内部結合を持つ場合には，宣言された型は不完全型にすることはできません。結合については，第 2.8 節を参照してください。

次に，仮定義の例を示します。

```
int i1 = 1;    /* Standard definition with external linkage */
int i4;        /* Tentative definition with external linkage */
static int i5; /* Tentative definition with internal linkage */
int i1;        /* Valid tentative definition, refers to previous */
               /* i1 declaration                               */
```

2.10 記憶域クラス

記憶域クラスは，データ・オブジェクトと関数の仮引数のみに適用されます。ただし，Compaq C の記憶域クラス・キーワードは関数の可視性に影響を与えるためにも使用します。プログラムで使用する各データ・オブジェクトと仮引数は，それぞれ 1 つだけ記憶域クラスを持ちます。記憶域クラスは明示的にも，特に指定しない場合でも割り当てられます。記憶域クラスには次の 4 種類があります。

- auto
- register
- static
- extern

オブジェクトの記憶域クラスは、リンカに対する有効性とその記憶域存続期間を決定します。外部結合または内部結合を持つオブジェクト、あるいは static 記憶域クラス指定子を持つオブジェクトは、静的記憶域存続期間を持ちます。この期間は、main が実行を開始する前にオブジェクトの記憶域が確保され、1 回だけ 0 に初期化されます。結合および static 記憶域クラス指定子を持たないオブジェクトは、自動記憶域存続期間を持ちます。このオブジェクトの場合には、記憶域はそれが宣言されたブロックに入ると自動的に割り当てられ、ブロックから出ると自動的に割当てが解除されます。自動オブジェクトは初期化されません。

関数に適用する場合には、extern 記憶域クラス指定子によって、関数を他のコンパイル単位から可視状態にします。また、static 記憶域クラス指定子によって、関数を同じコンパイル単位内の別の関数に対してのみ可視状態にします。次にその例を示します。

```
static int tree(void);
```

これ以降の項では、それぞれの記憶域クラスについて説明します。

2.10.1 auto クラス

auto クラスは、オブジェクトを定義するブロックの開始時にオブジェクトの記憶域を作成し、ブロックの終了時に破壊するように指定します。このクラスは、ブロック (例: 関数本体) の始まりでのみ宣言することができます。次にその例を示します。

```
auto int a;          /* Illegal -- auto must be within a block */
main ()
{
    auto int b;       /* Valid auto declaration */
    for (b = 0; b < 10; b++)
    {
        auto int a = b + a; /* Valid inner block declaration */
    }
}
```

autoオブジェクト (第 4.2 節を参照のこと) を持つ初期化子を使用すると、そのオブジェクトは作成するたびに初期化されます。ブロックの通常処理またはブロックへの飛越し文のいずれかにより、そのオブジェクトを含むブロックに入ってくると、記憶域はそのオブジェクト用に確保されます。ただし、飛越し文でブロックに入ると、そのオブジェクトが初期化されない場合があります。オブジェクトが可変長配列である場合、記憶域は予約されません。

autoクラスは、ブロック・スコープを持つオブジェクトの省略時の設定です。autoクラスを持つオブジェクトは、リンカで使用できません。

注意

ネストのブロックに入ると、その外側のブロックの実行は中断されますが終了はしません。ブロック内から関数を呼び出すと、その呼出しを含むブロックの実行は中断されますが終了はしません。このような場合には、確保された記憶域を持つ自動オブジェクトは、その記憶域を保存したままです。

2.10.2 register クラス

registerクラスは割り当てられたオブジェクトの使用頻度が高いかどうかを識別し、高い場合にはアクセス時間を最小にするためにそのオブジェクトをレジスタに割り当てるようにコンパイラに通知します。register は省略時のクラスとしては設定できず、明示的に指定しなければなりません。

registerクラスは autoクラスと同じ記憶域存続期間を持ちます。つまり、記憶域はオブジェクトを定義するブロックの開始時に registerオブジェクト用に作成され、そのブロックの終了時に破壊されます。

registerクラスは、関数の仮引数に明示的に指定できる唯一の記憶域クラスです。

*Compaq C*コンパイラは高度なレジスタ割当て技法を使用しているため、register キーワードは必要ありません。

2.10.3 static クラス

staticクラスは、そのプログラムの開始から終了までの間、識別子のための領域を保存するように指定します。静的オブジェクトはリンカには使用できません。したがって、別のコンパイル単位に異なるオブジェクトを参照する同一の宣言を含めることができます。

静的オブジェクトは、プログラム中で宣言を指定できる場所であればどこにでも宣言することができます。つまり、autoクラスのようにブロックの開始時に宣言する必要はありません。データ・オブジェクトを関数外で宣言すると、省略時の静的記憶域存続期間が与えられ、プログラムの開始時に 1 回だけ初期化されます。

staticオブジェクトを初期化するために使用する式は、定数式でなければなりません。static記憶域存続期間を持つオブジェクトが明示的に初期化されない場合には、そのオブジェクトの各算術メンバは 0 に初期化され、各ポインタ・メンバは空ポインタ定数に初期化されます。さまざまなデータ型のオブジェクトを初期化する方法については、第 4.2 節を参照してください。

2.10.4 extern クラス

externクラスは、ファイル・スコープを持つオブジェクトの省略時のクラスです。関数の外にあるオブジェクト (外部定義) は、宣言で明示的に staticキーワードが割り当てられない限り externクラス記憶域を持ちます。externクラスは staticオブジェクトと同じ記憶域存続期間を指定しますが、オブジェクト名または関数名はリンカからは隠されません。宣言に externキーワードを使用するとほとんどの場合に外部結合となり (第 2.8 節を参照のこと)、オブジェクトは静的記憶域存続期間を持ちます。

2.11 記憶域クラス修飾子

Compaq C には、次のような記憶域クラス修飾子があります。

```
__inline  
__forceinline  
__align
```

`inline`

リストの上から3番目までは、すべてのプラットフォームのすべてコンパイラ・モードで、有効なキーワードとして識別されます。これらは、Cの処理系用に予約されたネームスペースにあり、そのためこれらをユーザ宣言識別子として扱う必要はありません。これらはすべてのプラットフォームで同一の効果を持ちますが、*OpenVMS VAX*システムでは、`__forceinline`修飾子は`__inline`修飾子以上のインライン処理を行いません。

`inline`記憶域クラス修飾子は、`relaxed ANSI C`モードの場合か、`/ACCEPT=C99_KEYWORDS` (*OpenVMS*)または`/ACCEPT=GCCINLINE` (*OpenVMS*)修飾子を指定した場合にサポートされます。

注意

*Compaq C for OpenVMS Systems*は、*VAX C*キーワードとして、記憶域クラス修飾子の`noshare`、`readonly`、`align`のサポートも行っています。これらの記憶域クラス修飾子についての詳細は、*Compaq C for OpenVMS システム ユーザーズ・ガイド (OpenVMS)*を参照してください。

記憶域クラス指定子および記憶域クラス修飾子は、任意の順序で使用できます。通常、ソース・コードでは修飾子は指定子の後に置きます。次の例を参照してください。

```
extern noshare int x;
/* Or, equivalently ... */
int noshare extern x;
```

ただし、記憶域クラス指定子を最初以外の任意の位置に置く方式は使用されなくなっています。

以降の各項では、各 *Compaq C* 記憶域クラス修飾子について説明します。

2.11.1 `__inline` 修飾子

`__inline`記憶域クラス修飾子は、インライン拡張用の関数を指定します。関数定義およびプロトタイプに`__inline`を使用することにより、コンパイラがその関数の呼出し毎にその関数定義内のコードを置換するように指定します。置換はコンパイラの判断に応じて発生します。`__inline`記憶域クラス修飾子は、`#pragma inline`プリプロセッサ指示文と同一の効果を持ちます。ただし、`#pragma inline`は、選択された関数だけでなく、変換ユニット内のすべての関数についてインライン拡張を提供しようとしています(`#pragma inline`についての詳細は、プラットフォーム固有の *Compaq C* ドキュメントを参照してください)。

関数にインライン拡張を割り当てる場合は、次の書式を使用します。

```
__inline [型] function_definition
```

/STANDARD=PORTABLE モードで`__inline`が使用された場合、これは処理系固有の拡張であるため、コンパイラは警告を通知します。

次は、`__inline`を使用する例です。

```
/* prototype */
__inline int x (float y);

/* definition */
__inline int x (float y)
{
    return (1.0);
}
```

2.11.2 `inline` 修飾子

`__inline`記憶域クラス修飾子と同じように、`inline`記憶域クラス修飾子は関数宣言の宣言指定子として使用できます。

`inline`記憶域クラス修飾子は、`relaxed ANSI C` モードの場合か、`/ACCEPT=C99_KEYWORDS (OpenVMS)`または`/ACCEPT=GCCINLINE (OpenVMS)`修飾子が指定された場合にサポートされます。

静的関数では、`inline`を使用すると、`__inline`または`#pragma inline`を関数へ適用した場合と同じ効果があります。

ただし、外部結合の関数に`inline`を適用した場合、その翻訳単位内の呼出しがインライン化されるのに加えて、`inline`の意味に、他の翻訳単位内でもその関数への呼出しがインライン化されるか、その関数は外部関数として呼び出されるかはコンパイラが判断できるという規則が追加されました。

- `inline`キーワードが外部結合の関数宣言で使用された場合、その関数は同じ翻訳単位内でも定義されていなければならない。
- 関数のすべてのファイル・スコープ宣言が、`inline`キーワードを使用し、`extern`キーワードを使用しない場合、その翻訳単位内の定義はインライン補助定義と呼ばれ、外部呼出しが可能な(グローバル)定義はそのコンパイル単位内では生成されない。

上記の条件以外の場合、外部呼出しが可能な定義が、コンパイル単位内で生成される。

- `inline`補助定義は、静的記憶域存続期間を持つ、変更可能なオブジェクトの定義を含んではならない。また、内部結合の識別子を参照してはならない。これらの制限は、外部呼出しが可能な定義には適用されない。
- 通常、ある関数の外部呼出し可能な定義を持つコンパイル単位は、プログラム全体の中で1つまでである。
- 外部結合の関数への呼出しはすべて、`inline`修飾子の有無にかかわらず、外部関数への呼出しに翻訳されることがある。この事項と前項から、外部結合の関数が呼び出される場合、その関数の外部呼出し可能な定義は、プログラム全体のすべてのコンパイル単位の中で1つだけでなければならない。
- 外部結合の`inline`関数のアドレスは、一意に定まる外部呼出し可能な定義のアドレスとして必ず算出され、インライン定義のアドレスになることはない。

- アドレスがポインタに格納されている関数の名前をコンパイラが判断できる場合、外部呼出しが可能な定義へのポインタによるinline関数呼出しは、インラインのままであることも、インライン定義への呼出しとして翻訳されることもある。
- inlineキーワードがない場合、ヘッダ・ファイルが複数の翻訳単位にインクルードされていれば、ヘッダ・ファイル内の関数定義はリンク時に MULDEF エラーになる。このような関数定義では、MULDEF エラーを回避する方法の1つとしてinlineを指定することができる。例 (第 2.11.2.1 項) を参照。

注意

ここでは、C99 規格のinlineキーワードの意味について説明します。

gcc コンパイラでは、外部結合関数用として、この C99 インライン機能と同じような機能を備えたインライン関数宣言指定子を実装していますが、詳細な使用方法はやや異なっています。特に、externキーワードなしでinlineキーワードを使用するのではなく、externキーワードとinlineキーワードを組み合わせるとインライン定義になります。

/ACCEPT=[NO]GCCINLINE 修飾子は、機能のどのバリエーションが実装されているかを制御します。

2.11.2.1 例 — インライン関数指定子の使用

次のような C コードがあるとします。このコードでは、関数識別子 (my_max) が多重定義になります。

```
$ type t.h
int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
$
$ type a.c
#include "t.h"
```

```
main()
{
    int a =1;
    int b=2;

    func1();
    my_max(func1(a,b),20);
}
$
$ type b.c
#include "t.h"

void func1(int p1, int p2)
{
    my_max(p1,p2);
}
$
$ link a,b
%LINK-W-MULDEF, symbol MY_MAX multiply defined
      in module B file DISK$:[TEST.TMP]B.OBJ;4
```

この問題を回避する方法の1つとして、my_max関数をstaticキーワードで定義する方法があります。

```
static int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
```

ただしこの場合、グローバルに認識されるmy_max関数があるのではなく、各モジュールにmy_maxのコピーがあり、コピーのアドレスはそれぞれ異なっています。このため、関数ポインタの比較はうまくいきません。

ISO C99 では、この問題をinlineキーワードで解決しています。次のようにinlineをヘッダ・ファイルt.hに追加すると、MULDEF エラーを回避できます。

基本事項

2.11 記憶域クラス修飾子

```
inline int my_max (int x, int y)
{
    if (x >= y)
        return (x);
    else
        return (y);
}
```

このタイプの関数定義は、`__inline`キーワードを指定した場合と同じように、コンパイラによってインライン化される可能性があることを示します。異なるのは、`inline`関数ではコンパイラが、宣言される関数(この例では`my_max`)に対応する、関数のインライン補助定義を作成するという点です。ここで、コンパイラは次のいずれかを行います。

1. 補助関数を呼び出す。
2. グローバル関数(`my_max`)を呼び出す。この場合、アプリケーションのいずれかのモジュールに、非`static inline`関数のグローバル定義が必ずなければならない。
3. `my_max`への呼出しのインライン・コードを生成する。

1つの`inline`関数に対して作成できるグローバル定義は、アプリケーション内で1つだけです。インライン補助定義はモジュールごとに1つ、または補助関数のプロトタイプ宣言をモジュールごとに複数作成できます。

モジュールのいずれか(この例では`a.c`)に、次のようにファイル・スコープ関数宣言をインクルードすることで、グローバル・インライン定義を作成できます。

1. `inline`キーワードを除去する。

```
#include "t.h"
int my_max (int x, int y);
```

または

```
#include "t.h"
extern int my_max (int x, int y);
```

2. または、`inline`キーワードに`extern`記憶域クラスを指定する。

```
#include "t.h"
extern inline int my_max (int x, int y);
```

注意

`inline`関数のアドレスを取得すると、補助関数ではなく、必ずグローバル関数のアドレスとなります。

2.11.3 `__forceinline` 修飾子

`__inline`記憶域クラス修飾子と同様、`__forceinline`記憶域クラス修飾子は関数にインライン拡張を指定します。ただし、関数定義およびプロトタイプに`__forceinline`を使用した場合は、その関数へのすべての呼出しに対して関数定義内のコードを置換しなければならないことをコンパイラに指示します(`__inline`を使用した場合は、置換はコンパイラの自由な判断に応じて発生します)。

*OpenVMS VAX*システムでは、`__forceinline`記憶域クラス修飾子は`__inline`修飾子が行う以上のインライン処理を発生させません。

関数に強制的なインライン拡張を指定する場合は、次の書式を使用します。

```
__forceinline [型] function_definition
```

`/STANDARD=PORTABLE` モードで`__forceinline`が使用された場合、これは処理系固有の拡張であるため、コンパイラは警告を通知します。

2.11.4 `__align` 修飾子

`__align`と`_align`記憶域クラス修飾子の意味は、同じです。異なるのは、`__align`はすべてのコンパイラ・モードで使用できるキーワードで、`_align`はVAX Cキーワードを認識するモードのみで使用できるキーワードであるという点です。新しいプログラムでは、`__align`を使用してください。

`__align`記憶域クラス修飾子は、*Compaq C* データ型のオブジェクトを、指定された記憶域境界上に調整します。データ宣言やデータ定義では、`__align`修飾子を使用してください。

たとえば、整数を次クォードワード境界上に調整するには、次の宣言のいずれかを使用します。

```
int __align( QUADWORD ) data;  
int __align( quadword ) data;  
int __align( 3 ) data;
```

データの境界調整の位置を指定するときは、定義済み定数を使用するか、2の累乗を示す整数値を指定することができます。これらの定数や、明示的な2の累乗は、データを境界調整するときに埋め込むバイト数を *Compaq C* に指示します。前の例では、`int __align(3)` で 2^3 バイト(8バイト、クォードワード・メモリ)の境界調整を指定しています。

表 2-1 では、すべての定義済み境界調整定数とそれに対応する2の累乗数、バイト数を示しています。OpenVMS VAXシステムの場合は、定数0、1、2、3、4、または9を指定することができます。OpenVMS Alphaシステムの場合は、0～16の任意の定数を指定できます。

表 2-1 定義済み境界調整定数

定数	2の累乗数	バイト数
BYTE または byte	0	1
WORD または word	1	2
LONGWORD また は longword	2	4
QUADWORD また は quadword	3	8

(次ページに続く)

表 2-1 (続き) 定義済み境界調整定数

定数	2 の累乗数	バイト数
OCTAWORD または octaword	4	16
	5 (<i>Alpha only</i>)	32
	6 (<i>Alpha only</i>)	64
	7 (<i>Alpha only</i>)	128
	8 (<i>Alpha only</i>)	256
	9	512
	10 (<i>Alpha only</i>)	1024
	11 (<i>Alpha only</i>)	2048
	12 (<i>Alpha only</i>)	4096
	13 (<i>Alpha only</i>)	8192
	14 (<i>Alpha only</i>)	16384
PAGE または page	15 (<i>Alpha only</i>)	32768
	16 (<i>Alpha only</i>)	65,536 (<i>Alpha only</i>)
	9 (<i>VAX only</i>)	512 (<i>VAX only</i>)

2.12 順方向参照

識別子はいったん宣言しておくとい自由に使用できます。識別子を宣言の前に使用するとその宣言は順方向参照と呼ばれ、次の場合以外にはエラーになります。

- 文ラベルの宣言前に、goto文が文ラベルを参照する。
- 構造体タグ、共用体タグ、または列挙型タグを宣言する前に使用する。

次に、順方向参照の有効な例と無効な例を示します。

基本事項

2.12 順方向参照

```
int a;
main ()
{
    int b = c;           /* Forward reference to c -- illegal      */
    int c = 10;
    glop x = 1;          /* Forward reference to glop type -- illegal */
    typedef int glop;
    goto test;           /* Forward reference to statement label --
                           legal                                     */
test:
    if (a > 0 ) b = TRUE;
}
```

次の例では、順方向参照で構造体タグを使用しています。

```
struct s
{ struct t *pt };        /* Forward reference to structure t      */
.                         /* (Note that the reference is preceded  */
.                         /* by the struct keyword to resolve      */
.                         /* potential ambiguity)                  */
struct t
{ struct s *ps };

```

2.13 タグ

タグは、プログラム内にある構造体、共用体、列挙型を参照する手段としてそれらと一緒に使用することができます。構造体、共用体、または列挙型の宣言にタグを含めると、その宣言が参照できるところにある場合はそれらを指定できます。

以下に、構造体タグ、共用体タグ、列挙型タグの使用例を示します。

```
struct tnode {           /* Initial declaration --          */
                           /* tnode is the structure tag      */
    int count;
    struct tnode *left, *right; /* tnode's members referring to tnode */
    union datanode *p;        /* forward reference to union type is
                               declared below          */
};
```



```

union datanode {                                /* Initial declaration -- */
                                                /* datanode is the union tag */
    int ival;
    float fval;
    char *cval;
} q = {5};

enum color { red, blue, green }; /* Initial declaration -- */
.                                /* color is the enumeration tag */
.
.
struct tnode x;                                /* tnode tag is used to declare x */
enum color z = blue;                          /* color tag declares z to be of
                                                type color; z is also
                                                initialized to blue */

```

この例で示すように、いったんタグを宣言しておくとおブジェクトを再定義しなくてもそのタグを使用して同じスコープ内の他の構造体、共用体、または列挙型の宣言を参照できます。

構造体または共用体を完全に宣言する前にタグを宣言した場合には、不完全型を作成することができます。不完全型はオブジェクト・サイズを指定しません。したがって、不完全型を生成するタグは、オブジェクト・サイズが不要の場合にのみ使用できます。この型を完了するには、同じスコープ内でそのタグを別に宣言することでオブジェクトを完全に定義しなければなりません。次の例では、s構造体型の不完全な宣言を後続の定義がどのように完了するかを示しています。

```

struct s;                                /* Tag s used in incomplete type declaration */
struct t {
    struct s *p;
};
struct s { int i; }; /* struct s definition completed */

```

不完全型の概念については、第 2.6 節で説明しています。

次の宣言を例にとります。

```

struct tag;
union tag;

```

この宣言は構造体型または共用体型を指定し、宣言のスコープ内でのみ見えるタグを宣言します。宣言は、スコープがあればその範囲内に同じタグを持つ他の型とは異なる新しい型を指定します。

以下に、相互に参照する2つの構造体を指定する場合の最初のタグ宣言を示します。

```
struct s1 { struct s2 *s2p; /*...*/ }; /* D1 */
struct s2 { struct s1 *s1p; /*...*/ }; /* D2 */
```

s2 がネストのスコープ内のタグとして宣言された場合には、D1宣言は s2 を参照し、D2 に宣言された s2タグは参照しません。この文脈依存をなくすために、D1の前に次の宣言を挿入できます。

```
struct s2;
```

これは、内側のスコープ内で新しい s2タグを宣言します。その後、D2 が型の指定を完了します。

2.14 左辺値と右辺値

右辺値は2、 $x+3$ 、または $(x + y) * (a - b)$ などの式の値です。右辺値は記憶域には割り当てられません。次のコードは、右辺値が0と1の場合の例を示します。

```
if (x > 0)
{
    y += 1;
}
x = *y;          /* The value pointed to by y is assigned to x */
```

x および y識別子は、割り当てられた記憶域を持つオブジェクトです。y へのポインタは左辺値を保持します。

左辺値は、そのプログラムで使用するオブジェクトの記憶位置を示す式です。オブジェクトの位置は左辺値に、そこに記憶させる値は右辺値に示されます。次の演算子は、常に左辺値を生成します。

```
[]  
*  
->
```

ドット演算子(.)は通常、左辺値を生成します。しかし、必ずしも左辺値を生成するわけではありません。たとえば、`f().m` は左辺値ではありません。

変更可能な左辺値とは、配列型、不完全型、または `const` 修飾型を持たない左辺値を示します。構造体が共用体の場合には、`const` 修飾型のメンバを持たない左辺値を示します。

2.15 名前空間

名前空間とは、識別子をプログラム内で使用したときの状況に基づいて行われる識別子分類を示します。名前空間により、同一の識別子がオブジェクト、文ラベル、構造体タグ、共用体メンバ、および列挙定数と同時に使用できるようになります。2つの異なる要素に同じスコープの識別子を同時に明瞭に使用することは、同一の識別子が異なる名前空間にある場合にのみ可能です。識別子の使用状況によって、同じ名前の要素が複数あるために起こる不明瞭さが解決されます。

名前空間には、次の4種類があります。

- 文ラベル
- 構造体タグ、共用体タグ、および列挙型タグ
- 構造体および共用体メンバの各セット
- 他の識別子(変数、関数、型定義、および列挙定数)

たとえば、`flower` 識別子を1つのブロック内で使用して変数および列挙型タグの両方を表すことができます。これは、変数およびタグが異なる名前空間にあるためです。その結果、内部ブロックは `flower` 列挙型タグに影響を与えずに、`flower` 変数を再定義できます。したがって、いろいろな目的のために同じ識別子を使用する場合には、その識別子を扱う名前空間およびスコープの規則を分析する必要があります。第2.3節では、スコープの規則について説明しています。

構造体、共用体、および列挙型メンバ名は、これらのオブジェクトに同時に共通して使用できます。メンバの参照にこれらを使用することで、識別子の意味の不明瞭さが解決されます。しかし、構造体、共用体、または列挙型タグは固有なものにしなければなりません。これは、この3つのオブジェクト型のタグが同じ名前空間を共有しているからです。

2.16 前処理

Compaq C プログラムの翻訳には、複数の段階があります。コンパイラを起動すると通常、実際のコンパイラ起動の前に以下の事象が発生します。

1. 3文字表記 (存在する場合) は単一文字の内部表現に置き換えられる。
2. 改行文字の直前にバックスラッシュ文字があると、その改行文字は削除され、次の行が連結されて1つの論理行が生成される。
3. ソース・ファイルは、前処理トークンと空白文字のシーケンスに分けられる。各コメントは1つの空白文字に置き換えられる。
4. 前処理命令が実行され、前処理マクロが展開される。`#include`前処理命令に指定されたファイルは、これまでの4つの手順を繰り返して処理される。
5. 各ソース文字集合のメンバおよび文字定数と文字列リテラル中の各エスケープ・シーケンスは、実行文字集合のメンバに変換される。
6. 隣接する文字列リテラル・トークンが連結され、隣接するワイド文字列リテラル・トークンが連結される。
7. 結果として生じたトークンの文字列が分析され、翻訳される。
8. リンク段階では、外部オブジェクトおよび関数の参照がすべて解決される。現在のコンパイル単位では定義されていない関数とオブジェクトの外部参照を解決するために、ライブラリ構成要素がリンクされる。このリンク出力は、すべてプログラム・イメージに収集される。

前述の4番目の手順は前処理と呼ばれ、コンパイラの別の単位で処理されます。各前処理命令は `#` 記号で始まる行で使用されます。`#` 記号の前に空白が入ることもあります。この行は、Compaq C ソース・ファイルの他の部分とは構文上独立して

おり、ソース・ファイルのどこにでも記述することができます。前処理命令の行は、論理行の終わりで終了します。

実際にプログラムをコンパイルせずに、ソース・ファイルの前処理を行うことができます (使用可能なコンパイラ・オプションについては、プラットフォームに固有の Compaq C のマニュアルを参照してください)。前処理命令については、第 8 章で説明しています。

2.17 型名

場合によっては識別子なしで型名を指定できるものや、識別子なしで型名を指定しなければならないものがあります。たとえば、関数のプロトタイプ宣言において関数の仮引数は型名でのみ宣言できます。また、ある型から別の型にオブジェクトをキャストする場合には、関連する識別子のない型名が必要です (キャストについては第 6.4.6 項を、関数プロトタイプについては第 5.5 節を参照してください)。キャストは、型名を使用して行います。型名とは、関数の宣言または識別子を省略した関数またはオブジェクトの宣言です。

表 2-2 に、型名とそれに関連した型の例を示します。

表 2-2 型名の例

構成	型名
<code>int</code>	<code>int</code>
<code>int *</code>	<code>int</code> へのポインタ
<code>int *[3]</code>	<code>int</code> への 3 つのポインタの配列
<code>int (*)[3]</code>	3 つの <code>int</code> の配列へのポインタ
<code>int *()</code>	<code>int</code> へのポインタを返す仮引数指定がない関数
<code>int (*) (void)</code>	<code>int</code> を返す仮引数がない関数へのポインタ
<code>int (*const []) (unsigned int , ...)</code>	未指定の数を持つ関数への <code>const</code> ポインタの配列。各関数は、 <code>unsigned int</code> 型の 1 つの仮引数と未指定の数の仮引数を持ち <code>int</code> を返す。

表 2-2 では、抽象宣言子の例も示しています。抽象宣言子とは識別子を持たない宣言子です。この例では、`int` 型名に続く文字が抽象宣言子を形成しています。

基本事項
2.17 型名

* , [] , および () 文字はすべて , 特定の識別子を指定せずに宣言子を示しています。

*Compaq C*ではデータ・オブジェクトの型により、オブジェクトが表現できる値の範囲と種類、オブジェクトのために確保されるマシン記憶域のサイズ、およびオブジェクトで実行できる演算が決定されます。また、関数にも型があり、関数の宣言に関数の返却値の型と仮引数型を指定することができます。

この章の各節では、次の内容について説明します。

- データ・サイズ (第 3.1 節)
- 汎整数型 (第 3.2 節)
- 浮動小数点型 (第 3.3 節)
- 派生型 (第 3.4 節)

次は派生型の種類です。

- 関数型 (第 3.4.1 項)
- ポインタ型 (第 3.4.2 項)
- 配列型 (第 3.4.3 項)
- 構造体型 (第 3.4.4 項)
- 共用体型 (第 3.4.5 項)
- void 型 (第 3.5 節)
- 列挙型 (第 3.6 節)
- 型修飾子 (第 3.7 節)
- 型定義 (第 3.8 節)

データ型

どんな言語においても，指定したオブジェクトまたは関数のデータ型の選択は基本的なプログラミング・ステップの1つです。プログラム中の各データ・オブジェクトまたは関数にはデータ型を指定する必要があり，明示的に割り当てるかまたは省略時の設定により割り当てられます (オブジェクトへのデータ型の割当てについては，第4章を参照してください)。Compaq Cはさまざまなデータ型を提供しています。この多様性がCompaq Cの強力な機能ですが，慣れないうちは混乱するかもしれません。

混乱を避けるために，Compaq Cの基本型は数種類と考えてください。それ以外の型はこの基本型を組み合わせたものです。それらのいくつかは複数の方法で指定できます。たとえば，short と short int は同じ型です (本書では，最も明瞭な最長の名前を使用します)。型は，宣言の一部として各オブジェクトまたは関数に割り当てます。宣言についての詳細は，第4章を参照してください。

表 3-1 は，基本データ型の一覧です。基本データ型には次の種類があります。

- 汎整数型 — 特定範囲内の整数を表すオブジェクト
- 浮動小数点型 — 有効数字部 (整数部と小数部) とオプションの指数部で数字を表すオブジェクト
- 文字型 — 印字可能な文字を表すオブジェクト

文字型は整数として格納されます。

注意

列挙型は通常，汎整数型として分類されますが，この一覧には基本データ型しか示していません。詳細については，第3.6節を参照してください。

表 3-1 基本データ型

汎整数型	浮動小数点型
short int	float
signed short int	double
unsigned short int	long double
int	float _Complex (Alpha only)
signed int	double _Complex (Alpha only)
unsigned int	long double _Complex (Alpha only)
_Imaginary	
<i>Compaq C</i> では、_Imaginary キーワードを使用すると警告が出力されます。ただしこの警告は、このキーワードを通常の識別子として扱うようにすると解決します。	
long int	
signed long int	
unsigned long int	
long long int (Alpha only)	
signed long long int (Alpha only)	
unsigned long long int (Alpha only)	
_Bool	
汎整数文字型	
char	
signed char	
unsigned char	

汎整数型と浮動小数点型をまとめて、算術型と呼びます。整数値と浮動小数点値のサイズと範囲については、第 3.1 節を参照してください。

基本型からはさまざまな派生型を作成できます。派生型については、第 3.4 節を参照してください。

データ型

基本型と派生型の他に、固有の型を指定するための `void`、`enum`、および `typedef` の 3 つのキーワードがあります。

- `void` キーワードは値がないことを示す特別な型を指定するか、またはポインタ演算子 (`*`) とともに使用して汎用ポインタ型を示すことができる。`void` の使用法については、第 3.5 節を参照のこと。
- `enum` キーワードは、プログラマが作成した整数型を指定する。この指定は、整数定数値の定義済みセットに対してその型の許容値を指定することにより行う。列挙型は整数として格納される。列挙型の詳細については第 3.6 節を参照のこと。
- `typedef` キーワードは、1 つ以上の基本型または派生型から構成される型の同義語を指定する。型定義の詳細については第 3.8 節を参照のこと。

さらに、型修飾子キーワードがあります。

- `const` はオブジェクトへの書き込みアクセスを防ぐために使用する (第 3.7.1 項を参照のこと)。
- `volatile` はオブジェクトの参照で行われる可能性がある最適化を禁止するために使用する (第 3.7.2 項を参照のこと)。
- `__unaligned` (*Alpha*) は、ポインタ定義で使用され、指されているデータが、正しいアドレスに適切に境界調整されていないことをコンパイラに示す。
- `__restrict` (ポインタ型の場合のみ) は、ポインタが明確なオブジェクトを指していることを示すために使用され、コンパイラによる最適化が行われるようにする (第 3.7.4 項を参照)。

オブジェクトの型宣言に修飾子キーワードを使用すると修飾型になります。型修飾子の詳細については、第 3.7 節を参照してください。

このようにさまざまな型が存在するために、プログラムでは異なる型のオブジェクト上での演算が必要になる場合があります。また、ある型の仮引数を別の仮引数型を返す関数に渡す必要が生じる場合もあります。*Compaq C* では異なる種類の変数を異なる方法で格納するため、最低 1 つのオペランドまたは実引数を変換して、そのオペランドまたは実引数の型をもう一方と一致させる必要があります。変換は明示的にキャストを使用して行うか、またはコンパイラを介して暗黙に行うことがで

きます。データ型の変換については第 6.11 節を，型の互換性については第 2.7 節を参照してください。

実行定義のデータ型については，プラットフォームに固有の *Compaq C* ドキュメントを参照してください。

3.1 データ・サイズ

指定したデータ型のオブジェクトは，個別のサイズを持つメモリのセクションに格納されます。異なるデータ型のオブジェクトには，それぞれ異なるメモリが必要です。表 3-2 は，基本データ型のサイズと範囲を示しています。

表 3-2 データ型のサイズと範囲

型	サイズ	範囲
汎整数型		
short int または signed short int	16 ビット	-32768 ~ 32767
unsigned short int	16 ビット	0 ~ 65535
int または signed int	32 ビット	-2147483648 ~ 2147483647
unsigned int	32 ビット	0 ~ 4294967295
long int または signed long int (<i>OpenVMS</i>)	32 ビット	-2147483648 ~ 2147483647
long int または signedlong int (<i>Tru64 UNIX</i>)	64 ビット	-9223372036854775808 ~ 9223372036854775807
unsigned long int (<i>OpenVMS</i>)	32 ビット	0 ~ 4294967295
unsigned long int (<i>Tru64 UNIX</i>)	64 ビット	0 ~ 18446744073709551615
signed long long int (<i>Alpha only</i>) , signed __int64 (<i>Alpha only</i>)	64 ビット	-9223372036854775808 ~ 9223372036854775807
unsigned long long int (<i>Alpha only</i>) , unsigned __int64 (<i>Alpha only</i>)	64 ビット	0 ~ 18446744073709551615

(次ページに続く)

データ型

3.1 データ・サイズ

表 3-2 (続き) データ型のサイズと範囲

型	サイズ	範囲
汎整数文字型		
char および signed char	8 ビット	-128 ~ 127
unsigned char	8 ビット	0 ~ 255
wchar_t	32 ビット	0 ~ 4294967295
浮動小数点型 (範囲は絶対値用)		
float	32 ビット	$1.1 \times 10^{-38} \sim 3.4 \times 10^{38}$
double	64 ビット	$2.2 \times 10^{-308} \sim 1.7 \times 10^{308}$
long double (<i>OpenVMS, Alpha</i>)	128 ビット	$3.4 \times 10^{-49321} \sim 1.2 \times 10^{1049321}$
long double (<i>OpenVMS VAX, Tru64 UNIX</i>)	double と同様	double と同様

派生型の場合は、さらにメモリ空間が必要になる可能性があります。

実行定義のデータ型のサイズについては、プラットフォームに固有の *Compaq C* ドキュメントを参照してください。

3.2 汎整数型

C 言語では、次の汎整数型を宣言することができます。

- 符号付き、または符号なしの整数値
- ブール値 (0 が偽に等しく、0 以外の数が真に等しい)
- 文字 (コンパイラによって自動的に整数値に変換される)
- 列挙型のメンバ (コンパイラによって整数として解釈される)
- ビット・フィールド

汎整数型は次のとおりです。

- char, signed char, unsigned char — 8 ビット

- short int, signed short int, unsigned short int — 16 ビット
- _Bool—1 バイト
- int, signed int, unsigned int — 32 ビット
- long int, signed long int, unsigned long int — 32 ビット (*OpenVMS*)
- long int, signed long int, unsigned long int — 64 ビット (*Tru64 UNIX*)
- signed long long int (*Alpha only*) , unsigned long long int (*Alpha only*) —64 ビット
- signed __int64 (*Alpha only*) , unsigned __int64 (*Alpha only*) —64 ビット
- enum — 32 ビット

3.2.1 非文字型

*OpenVMS*システム上の *Compaq C*では, int と long の記憶域は同一です。さらに, signed int と signed long の記憶域も同一であり, unsigned int と unsigned long の記憶域も同一です。

*Tru64 UNIX*システム上の *Compaq C*では, intデータ型の記憶域は 32 ビットであり, long intデータ型の記憶域は 64 ビットです。

64 ビットの汎整数型signed long long intおよびunsigned long long intと, これらと同等のsigned __int64およびunsigned __int64は, Alpha プロセッサ上でのみ提供されています。注意: __int64データ型とlong long intデータ型 (符号付きおよび符号なし) は, ポインタ演算の場合を除き, 取り替えて使用することができません。ポインタ演算の場合は, ポインタの型が同じでなければなりません。

```
__int64 *p1;  
__int64 *p2;  
long long int *p3;  
.  
.  
.  
p1 = p2;          // valid  
p1 = p3;          // invalid
```

データ型

3.2 汎整数型

符号付き汎整数型には、それぞれに対応する同じ記憶容量を使用する符号なし汎整数型が存在します。汎整数型に `unsigned` キーワードを指定すると、記憶域の正の値の範囲を広げることで整数値の解釈方法を変更できます。`unsigned` キーワードを使用すると、ビットは異なる方法で解釈され、符号なし型を持つ正の範囲を(負の範囲を狭めることによって) 広げることができます。次にその例を示します。

```
signed short int x = 45000; /* ERROR -- value too large for short int */
unsigned short int y = 45000; /* This value is OK */
```

`signed short int` 型の値の範囲は $-32,768 \sim 32,767$ です。`unsigned short int` 型の値の範囲は $0 \sim 65,535$ です。

符号なしのオペランドを含む計算は、オーバフローすることはありません。これは、`unsigned` 型の範囲外の結果は剰余算術の規則によってその型に合うように調整されるからです。結果を整数型で表せない場合には、その数が符号なし整数型で表すことができる最大値よりも大きいものとして結果を調整します。つまり、下位ビットは保持され、結果の型に収まらない数値結果の上位ビットは破棄されます。次にその例を示します。

```
unsigned short int z = (99 * 99999); /* Value of y after evaluation is 3965 */
```

Compaq C では、*VAX C* および他の *C* コンパイラとの互換性のため、省略時の設定では単純な `char` 型を `signed` として処理します。ただし、コマンド行オプションでこれを制御することが可能であり、事前に定義されているマクロをテストして、任意のコンパイルにおけるオプションの設定を判断することができます。Alpha システムでは、`unsigned char` を使用すると、文字を多用する処理の効率が向上する場合があります。

n ビットの符号なし整数は常に $0 \sim 2^n - 1$ の範囲の変数で、符号なしの 2 進表現に解釈されます。

注意

符号付き整数の解釈は、マシン表現のサイズとそのマシンで使用されるコード化の技法によって異なります。2 の補数表現の場合、 n ビットの符号付き整数の範囲は $-2^{n-1} \sim 2^{n-1} - 1$ になります。

C99 規定の `_Bool` データ型は、コンパイラの VAX C モード、コモン・モード、および strict ANSI89 モード以外のすべてのモードで利用できます。`_Bool` オブジェクトの記憶域は 1 バイトで、unsigned integer として扱われます。ただし、その値は 0 か 1 だけです。

注意

- ビット・フィールドは、`_Bool` 型として宣言できます。
- ポインタを `_Bool` 型に変換することができます。
- スカラ値を `_Bool` に変換する場合、値が 0 と等しければ (たとえば、ポインタが `NULL`)、結果は 0 です。値が 0 と等しくなければ、結果は 1 です。このことは、`_Bool` 型が他の整数型と違う点です。次の例では、`b` の値は 0 ですが、`c` の値は 1 です。

```
double a = .01;
int b = a;
_Bool c = a;
```

- `_Bool` 型は、新しい標準ヘッダ `<stdbool.h>` と一緒に使用することを前提としています (ただし、必須ではありません)。新しいヘッダの内容は、次のとおりです。

```
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

第 9.11 節も参照してください。

3.2.2 文字型

文字型は、`char` キーワードで宣言される汎整数型です。`char` オブジェクトは、非整数演算には使用しないようにしてください。非整数演算に使用すると移植できなくなることがあります。`char` 型として宣言するオブジェクトは、常にソース文字集合の最大メンバを格納することができます。

次に有効な文字型を示します。

- char
- signed char
- unsigned char
- wchar_t

wchar_tワイド文字型は、ASCII 文字集合に含まれていない文字を表現するために提供されています。wchar_t型は、<stddef.h>ヘッダ・ファイルに typedefキーワードを使用して定義されています。定数または文字列に使用するワイド文字の直前には、L を付ける必要があります。次にその例を示します。

```
#include <stddef.h>
wchar_t a[6] = L"Hello";
```

charオブジェクトはすべて 8 ビットで格納されます。wchar_tオブジェクトは unsigned intオブジェクトと同様に 32 ビットで格納されます。指定した文字の値は使用している文字集合で決定されます。本書に記載しているすべての例では ASCII 文字集合を使用しています。10 進基数、8 進基数、および 16 進基数に対応する ASCII 値の一覧については、付録 C を参照してください。

移植性を高めるために、算術に使用する charオブジェクトは signed char または unsigned char として宣言します。次にその例を示します。

```
signed char letter;
unsigned char symbol_1, symbol_2;
signed char alpha = 'A'; /* alpha is declared and initialized as 'A' */
```

文字列はヌル文字 '\0' で終了する文字の配列です。文字列使用の構文規則についての詳細は第 1.9.3 項を、文字列リテラルの宣言については第 4 章を参照してください。

3.3 浮動小数点型

浮動小数点型には，次の種類があります。

- float — 32 ビット
- double — 64 ビット
- long double (*OpenVMS ALPHA*) — 128 ビット (省略時の設定)，64 ビット (オプション)
- long double (*Tru64 UNIX*) — 64 ビット (*Tru64 UNIX*の最新バージョン)
- long double (*VAX*) — 64 ビット
- float _Complex (*Alpha only*)
- double _Complex (*Alpha only*)
- long double _Complex (*Alpha only*)

浮動小数点型は，小数部がある変数，定数，または関数の返却値に使用します。また，値が汎整数型で格納可能な範囲を超える場合にも浮動小数点型を使用します。次に，浮動小数点型宣言 (および初期化) の例を示します。

```
float x = 35.69;
double y = .0001;
double z = 77.0e+10;
float Q = 99.9e+99;           /* Exceeds allowable range */
```

3.3.1 複素数型 (*Alpha only*)

C99 規格では，Fortran の型に似た組み込み複素数データ型が，3 つのすべての精度 (float _Complex，double _Complex，および long double _Complex) で導入されました。C99 規格には，これに関連するヘッダ・ファイル <complex.h> もあります。<complex.h> ヘッダ・ファイルでは，“complex” というマクロを定義しています。これらの型を参照するときには，このマクロを使用することをお勧めします (第 9.2 節を参照)。

データ型

3.3 浮動小数点型

複素数型の表現および境界合わせの要件は、対応する実数型の要素を2つ含む配列型と同じです。1番目の要素は複素数の実数部と同じで、2番目の要素は虚数部と同じです。

この型の使用法は、Fortranの場合と似ています。定数には、特別な構文はありませんが、実数部が0で虚数部が1.0の複素数値を持つ新しいキーワード `_Complex_I` があります。ヘッダ・ファイルには、`_Complex_I` に展開されるマクロ `I` が定義されています。このため、実数部と虚数部が同じ2.0の複素数定数は、 $2.0 + 2.0*I$ と記述できます。

Compaq C の複素数型に関しては、次の注意事項があります。

- `/FLOAT=D_FLOAT` コマンド行オプションを使用する場合、複素数データ型は利用できません。これは、恒久的な制限事項です。
- `long double complex` 以外の複素数型および関数は、Version 7.3 より前の OpenVMS で利用できます。`long double complex` 型は、OpenVMS Version 7.3 で利用できます。
- `long double complex` 変数の初期化付き宣言を使用すると、コンパイラでマシン語リストを作成するときに、妥当性チェックのエラーが発生します。
- 従来より `cabs`、`cabsf`、および `cabsl` という関数は、2つの浮動小数点値を持つ構造体表現を使用して、`<math.h>` で宣言されています。この宣言は、複素数値を渡す呼び出し標準とは互換性がありません。正しく動作する `cabs` 関数にアクセスするには、`<math.h>` をインクルードする前に `<complex.h>` をインクルードしなければなりません。

3.3.2 虚数型 (*Alpha only*)

C99規格では、Annex G(参考)に規定されている「純虚数」型という実験的なオプションの機能とともに使用する型指定子として、`_Imaginary` キーワードが予約されています。*Compaq C* では、`_Imaginary` キーワードを使用すると警告が出力されます。この警告は、このキーワードを通常の識別子として扱うようにすると解決します。

3.4 派生型

C 言語の派生型には、次の 5 種類があります。

- 関数型
- ポインタ型
- 配列型
- 構造体型
- 共用体型

これ以降の各項では、この派生型の 5 種類について説明します。

派生型は、1 つ以上の基本型の組み合わせから形成されています。派生型を使用すれば、多くの新しい型を形成することができます。配列型と構造体型をまとめて、集成型と呼びます。集成型には共用体型は含まれませんが、共用体には集成体メンバを含めることが可能です。

3.4.1 関数型

関数型は指定された型の値を返す関数について記述します。関数が値を返さない場合には、次のように「void を返す関数」として宣言しなければなりません。

```
void function1 ();
```

次の例では、関数のデータ型は「int を返す関数」です。

```
int uppercase(int lc)
{
    int uc = lc + 0X20;
    return uc;
}
```

一般的な宣言については、第 4 章で説明します。関数については第 5 章で、その宣言、仮引数、および実引数の引渡しについて説明します。

3.4.2 ポインタ型

ポインタ型は、指定された型のオブジェクトのアドレスを表す値を記述します。ポインタは、それが示すオブジェクトのアドレスを参照する整数値として格納されます。ポインタ型は他の型から派生する型であり、ポインタの被参照型と呼ばれます。次にその例を示します。

```
int *p;           /* p is a pointer to an int type           */
double *q();      /* q is a function returning a pointer to an                */
                  /* object of type double                                     */
int (*r)[5];      /* r is a pointer to an array of five elements             */
                  /* (r holds the address to the first element of            */
                  /* the array)                                              */
const char s[6]; /* s is a const-qualified array of 6 elements          */
```

ポインタ自体はどんな記憶域クラスでも持つことができますが、ポインタに示されるオブジェクトは register 記憶域クラスを持つことはできません。また、ビット・フィールドにすることもできません。修飾済み互換型または未修飾の互換型へのポインタは、そのポインタが示す型と同じ表現および境界調整条件を持ちます。これは互換型の場合のみであり、他の型の場合にはあてはまりません。

`void *` は汎用的な「void へのポインタ型」を指定します。`void *` を使用すれば、任意の型のオブジェクトを指定することができます。これは主に、異なる型または不明の型 (関数プロトタイプなど) を持つオブジェクトのアドレスを示すためにポインタが必要な場合に役に立ちます。さらに、`void` へのポインタは別の型のポインタと変換することもでき、文字型のポインタと同じ表現と境界調整条件を持ちます。

アドレス 0 へのポインタを空ポインタと呼びます。並びの次のメンバを示す際にポインタを使用しますが、並びのメンバがそれ以上存在しないことを示す場合は空ポインタを使用します。ただし、空ポインタを `*` または添字指定演算子で逆参照した場合の結果は保証できません。

ポインタ宣言の構文については、第 4 章を参照してください。

3.4.3 配列型

配列型は、有効な完了型から形成することができます。配列型を完了するには、配列メンバの数と型が明示的または暗黙に指定されていなくてはなりません。メンバ型は、同じコンパイル単位または異なるコンパイル単位で完了できます。配列は、`void`型または関数型にすることはできません。これは、`void`型は完了することができず、関数型は記憶域を必要とするオブジェクト型ではないからです。

一般的に、配列はいくつかの同種類の値で演算を実行するために使用します。配列型のサイズは、配列のデータ型と要素数によって決定されます。配列の各要素は同じ型を持っています。たとえば、次の定義では4文字の配列を作成します。

```
char x[] = "Hi!" /* Declaring an array x */;
```

要素のそれぞれが、`char`オブジェクトのサイズ(8ビット)を持っています。配列のサイズは初期化の時点で決定されます。この例では、配列は3つの明示的な要素と1つのヌル文字を持っています。4つの8ビットの要素は、32ビットのサイズを持つ配列になります。

配列は連続してメモリに割り当てられ、空(つまりメンバを持たない状態)にすることはできません。1つの配列は1次元でしか作成できません。「2次元」の配列を作成するには、配列の配列を宣言してください。

サイズの不明な配列を宣言することができます。この宣言はサイズが指定されないため、不完全配列宣言と呼びます。次の例は不完全な宣言を示しています。

```
int x[];
```

この方法で宣言する配列サイズは、プログラムのどこかで指定しなければなりません(不完全配列と配列の初期化については、第4.7節を参照してください)。

文字列(文字列リテラル)は `char` または `wchar_t` 型の形式で格納され、ヌル文字 `'\0'` で終了します。

*Compaq C*では、1つの配列は1次元でしか作成できません。ただし、配列の配列を宣言することにより、多次元配列を作成することができます。これらの配列の要素は、右端の添字が最初に変更するように、増加方向のアドレスに格納されま

す。これは行優先順序と呼ばれるもので、車の走行距離計に似ています。たとえば、`int a[2][3]`;として宣言した2つの配列では、次の順序で要素が格納されます。

```
a[0][0], a[0][1], a[0][2], a[1][0], a[1][1], a[1][2]
```

3.4.4 構造体型

構造体型は、メンバと呼ばれる空以外のオブジェクトが連続して割り当てられている集合です。構造体を使用すれば、異質のデータをグループ化することができます。これはPascal言語のレコードと類似しています。配列とは異なり、構造体の要素は同じデータ型の要素である必要はありません。また、構造体の要素は名前でアクセスされ、添字ではアクセスされません。次の例では、`employee`構造体を宣言しています。また、`employee` という構造体型の2つの構造体変数 (`ed` および `mary`) を宣言しています。

```
struct employee { char name[30]; int age; int empnumber; };  
struct employee ed, mary;
```

構造体メンバには、`void`型や関数型のような不完全型でなければ、どんな型でも指定することができます。構造体にはそれ自体の型のオブジェクトへのポインタを含めることはできますが、それ自体の型のオブジェクトをメンバとして含めることはできません。このようなオブジェクトは、不完全型を持つことになります。次に無効な例を示します。

```
/* This is invalid. */  
struct employee {  
    char name[30];  
    struct employee div1;    /* This is invalid. */  
    int *f();                /* This is also invalid. */  
};
```

次に有効な例を示します。

```
struct employee {
    char name[30];
    struct employee *div1; /* Member can contain pointer to employee
                           structure. */
    int (*f)();           /* Pointer to a function returning int */
};
```

宣言された構造体メンバ名は、その構造体内では固有でなければなりません。ただし、ネストされた構造体またはネストされていない別の構造体、あるいは異なるオブジェクトを参照している名前空間ではそのメンバ名を使用することができます。次にその例を示します。

```
struct {
    int a;
    struct {
        int a; /* This 'a' refers to a different object
               than the previous 'a' */
    } nested;
};
```

拡張機能として、*Compaq C*の緩和モードでは、構造体または共用体でメンバの名前を指定せずに(無名メンバと呼ぶ)、ネストされた構造体または共用体を宣言することができます。この効果は、ネストされた無名の構造体または共用体のメンバを、構造体または共用体にネストすることなく、直接宣言したのと同じです。そのため、前の例で、内部のstruct宣言から識別子nestedを省略した場合、内部の構造体のメンバaと、それが属している構造体のメンバaとの間で矛盾が発生するため、エラーになります。これは、(構造体と共用体の両方に許可されていることを除き)C++言語の無名共用体機能と類似しており、またVAX C固有のvariant_structおよびvariant_union機能と類似しています。

構造体とその宣言については、第4章を参照してください。

コンパイラは、メンバ宣言の順に構造体メンバの記憶域を割り当てます。この順序では、後続のメンバが増加するメモリ・アドレスに割り当てられます。最初のメンバは、必ずその構造体自体の開始アドレスから始まります。後続のメンバは境界調整単位ごとに境界調整されますが、これは構造体のメンバ・サイズによって異なることがあります。構造体は、そのメンバが正しく境界調整されるようにパディング

(未使用のビット)を含んでいることがあります。この場合、その構造体のサイズは、すべてのメンバに必要な記憶容量に境界調整のために必要なパディング領域用の記憶容量を追加したものになります。構造体の境界調整および表現に関するプラットフォーム固有の情報については、ご使用のシステムの *Compaq C* のマニュアルを参照してください。

あるプラットフォーム上の構造体の境界調整と、他のプラットフォーム上の境界調整とを一致させるためには、プラグマを使用します。プラグマについての詳細は、第 B.29 節を参照してください。

3.4.5 共用体型

共用体型には、メモリの同じ位置に異なる型のオブジェクトを格納できます。それぞれ異なる共用体メンバは、プログラム中の同じ位置を異なる時期に占有できます。共用体の宣言には共用体のすべてのメンバを含めることができ、共用体が保持できるすべてのオブジェクト型も入れることができます。共用体には一度に 1 つのメンバしか入れることができません。共用体に他のメンバを続けて代入すると、その同じ記憶域にある既存のオブジェクトは上書きされます。

共用体には有効な識別子の名前を付けることができます。空の共用体を宣言することはできず、共用体にそれ自体のインスタンスを含めることもできません。共用体のメンバとして、`void` 型、関数型、または不完全型は指定できません。共用体には、これらの型の共用体へのポインタを含めることができます。

見方を変えると、共用体とは異なる時期に異なる型のオブジェクトを表すことのできる単一オブジェクトであるといえます。共用体によって、マシン依存の構成を使用せずに、プログラムの処理状況に応じて型とサイズを変更できるオブジェクトを使用することができます。別の言語では、この概念を可変レコードと呼んでいます。

共用体を定義するための構文は、構造体の構文と非常に類似しています。各共用体型の定義は、それぞれ固有の型を作成します。共用体内では、各共用体メンバ名は固有でなければなりません。しかし、ネストされた共用体やネストされていない他の共用体または名前空間では、同じメンバ名を使用できます。次にその例を示します。


```
union {  
    int a;  
    union foo {  
        int a; /* This 'a' refers to a different object  
                than the previous 'a' */  
    } nested;  
};
```

Compaq C の拡張機能である緩和モードでは，C++ 言語と同じように無名の共用体メンバが許されることに注意してください。

共用体のサイズは，その最大のメンバに境界調整条件に必要なパディングを加えた記憶容量になります。

一度共用体を定義すると，その共用体に宣言されているオブジェクトに値を代入することができます。次にその例を示します。

```
union name {  
    dvalue;  
    struct x { int value1; int value2; };  
    float fvalue;  
} alberta;  
alberta.dvalue = 3.141596; /* Assigns the value of pi to the union object */
```

この場合，alberta は double 値，struct 値，または float 値を保持できます。プログラマは，共用体に含まれているオブジェクトの現在の型を追跡する必要があります。代入式を使用して，共用体に保持される値の型を変更できます。

ある型の値を格納するために共用体を使用して，その値を別の型でアクセスした場合の処理結果は定義されていません。次にその例を示します。

```
/*  
    Assume that 'node' is a typedef_name for objects for which  
    information has been entered into a hash table;
```

データ型

3.4 派生型

```
/*
    'hash_entry' is a structure describing an entry in the hash table.
    The member 'hash_value' is a pointer to the relevant 'node'.
*/
typedef struct hash_entry
{
    struct hash_entry *next_hash_entry;
    node *hash_value;
    /* ... other information may be present ... */
} hash_entry;

extern hash_entry *hash_table [512];

/*
    'hash_pointer' is a union whose members are a pointer to a
    'node' and a structure containing three bit fields that
    overlay the pointer value. Only the second bit field is
    being used, to extract a value from the middle
    of the pointer to be used as an index into the hash table.
    Note that nine bits gives a range of values from 0 to 511;
    hence, the size of 'hash_table' above.
*/
typedef union
{
    node *node_pointer;
    struct
    {
        unsigned : 4;
        unsigned index : 9;
        unsigned :19;
    } bits;
} hash_pointer;
```

3.5 void 型

void型は、完了することができない不完全型です。

void型には、次に示す重要な3つの使用方法があります。

- 関数が値を返さないことを示す。
- 汎用ポインタ (任意の型のオブジェクトを指すことのできるもの) を示す。

- 実引数のない関数プロトタイプを指定する。

次の例では、仮引数を持たず、値を返さない関数を定義するために void を使用する方法を示します。

```
void message(void)
{
    printf ("Stop making sense!");
}
```

次の例では、1 番目と 2 番目の実引数として、任意のオブジェクトへのポインタを受け付ける関数の関数プロトタイプを示します。

```
void memcpy (void *dest, void *source, int length);
```

void型へのポインタは、文字型へのポインタと同じ表現および境界調整条件を持ちます。void *型は voidを基本にした派生型です。

また、void型をキャスト式を使用して明示的に値を破棄したり、または値を無視することもできます。

```
int tree(void);

void main()
{
    int i;

    for (; ; (void)tree()){...} /* void cast is valid */
    for (; (void)tree(); ;){...} /* void cast is NOT valid, because the */
                                /* value of the second expression in a */
                                /* for statement is used */
    for ((void)tree(); ;){...} /* void cast is valid */
}
```

void式は値を持たないため、値が必要な場合はいかなる状況であっても使用できません。

3.6 列挙型

列挙型は、定義済み並びからオブジェクトの許容値を指定するために使用します。並びの要素は列挙定数と呼ばれます。列挙型の主な使用法は、シンボル名を明示的に表すことです。したがって、値を整数値で表すことができるオブジェクトを意図しています。

列挙型のオブジェクトは `signed int` 型のオブジェクトとして解釈され、他の汎整数型のオブジェクトと互換性があります。

コンパイラは、各列挙定数に整数値を (0 から開始して) 自動的に代入します。次の例では、列挙定数の並びを持つ `background_color` 列挙型オブジェクトを宣言しています。

```
enum colors { black, red, blue, green, white } background_color;
```

このプログラムでは、後で `background_color` オブジェクトに値を代入できます。

```
background_color = white;
```

この例では、コンパイラが整数を自動的に代入します (`black = 0`, `red = 1`, `blue = 2`, `green = 3`, `white = 4`)。この代わりに、列挙型の定義中に明示的に値を代入することもできます。

```
enum colors { black = 5, red = 10, blue, green = 7, white = green+2 };
```

この場合、`black` は整数値 5、`red` は 10、`blue` は 11、`green` は 7、`white` は 9 に等しくなります。`blue` は直前の定数値 (`red`) に 1 を加えた値に等しく、次の `green` はこの番号の連続からは外れます。

ANSI C 規格では列挙型への代入については厳密ではないため、定義済み並びにない代入値でも何の警告もなしに受け付けられます。

3.7 型修飾子

型修飾子には、次の 4 つがあります。

- `const`
- `volatile`
- `__unaligned (axp)`
- `__restrict` (ポインタ型のみ)

型修飾子は ANSI C 規格によって導入されているものであり、コンパイラの最適化をある程度制御できるようにするものです。`const`および`volatile`型修飾子は、任意の型に適用することができます。`__restrict`型修飾子はポインタ型にのみ適用できます。

`__restrict`型修飾子は、1989 ANSI C 規格に含まれていないため、このキーワードには先頭に 2 つのアンダスコア (`_`) が付けられています。C 規格の次のバージョン (9X) では、この節の記述と同じ意味で、キーワード`restrict`が採用される予定です。

`const` を使用するとオブジェクトへの書込みアクセスを制御でき、そのオブジェクトを含む関数呼出しの間で起こり得る副作用を排除できます。これは、通常は副作用によりオブジェクトの記憶域が変更されるのに対して `const` がこの変更を禁止するからです。

他のプロセスまたはハードウェアで変更される可能性のあるオブジェクトには、`volatile` を使用してください。`volatile` を使用すると、オブジェクトの参照に関する最適化が行われなくなります。オブジェクトが `volatile` で修飾されている場合には、初期化されてから代入されるまでの間にそれが変更される可能性があるため最適化することはできません。

すべての関数の仮引数が 1 つの仮引数の型修飾を共有することはありません。次にその例を示します。

```
int f( const int a, int b)    /* a is const qualified; b is not */
```

データ型

3.7 型修飾子

配列識別子とともに型修飾子を使用すると配列の要素は修飾されますが、配列型自体は修飾されません。

次の宣言と式は、型修飾子が配列型または構造体型を変更する場合を示しています。

```
const struct s { int mem; } cs = { 1 };
struct s ncs;                      /* ncs is modifiable */
typedef int A[2][3];
const A a = {{4, 5, 6}, {7, 8, 9}}; /* array of array of const */
                                   /* int's */
int *pi;
const int *pci;

ncs = cs;                          /* Valid */
cs = ncs;                          /* Invalid, cs is const-qualified */
pi = &ncs.mem;                     /* Valid */
pi = &cs.mem;                       /* Violates type constraints for = operator */
pci = &cs.mem;                     /* Valid */
pi = a[0];                         /* Invalid; a[0] has type "const int *"
```

3.7.1 const 型修飾子

値を変更できないオブジェクトを修飾するには、const型修飾子を使用します。constキーワードで修飾したオブジェクトは、変更することはできません。つまり、constとして宣言されているオブジェクトは、その値を変更する演算時にオペランドとしての役割を果たすことはできません。たとえば、++ および --演算子は const で修飾されるオブジェクトでは使用できません。オブジェクトに const修飾子を使用すると、記憶域を変更する操作によって引き起こされる副作用を防ぐことができます。

const を修飾したオブジェクトの宣言は、非修飾型のものよりも多少複雑になる可能性があります。次に、いくつかの例とその説明を示します。

```
const int x = 44; /* const qualification of int type.
                  The value of x cannot be modified. */
const int *z;     /* Pointer to a constant integer.
                  The value in the location pointed
                  to by z cannot be modified. */
int * const ptr;  /* A constant pointer, a pointer
                  that will always point to the
                  same location */
const int *const p; /* A constant pointer to a constant
                     integer: neither the pointer or
                     the integer can be modified. */
const const int y; /* Illegal - redundant use of const */
```

次は，const型修飾子に適用される規則です。

- const修飾子を使用して，構造体または共用体の単一メンバを含む任意のデータ型を修飾できる。
- 集成型を宣言するときに const を指定すると，集成型のすべてのメンバは constで修飾されるオブジェクトとして処理される。const を使用して集成型の1つのメンバを修飾すると，そのメンバのみが修飾される。次にその例を示す。

```
const struct employee {
    char *name;
    int  birthdate; /* name, birthdate, job_code, and salary are */
    int  job_code; /* treated as though declared with const.   */
    float salary;
    } a, b; /* All members of a and b are const-qualified*/
struct employee2 {
    char *name;
    const int birthdate; /* Only this member is qualified */
    int job_code;
    float salary;
    } c, d;
```

前の構造体のメンバは，すべて const で修飾される。プログラムの後の部分で別の構造体を指定するために employeeタグを使用すると，特に指定しない限り const修飾子は新しい構造体メンバに適用されない。

- `const`修飾子は `volatile`修飾子と一緒に指定できる。これは、ソース・プロセスでは変わらないが、他のプロセスでは変更可能であるデータ・オブジェクトの宣言に役立つ。また、実時間クロックなどのメモリ・マップ入力ポートのモデルとしても役立つ。
- 非`const`オブジェクトのアドレスは、(明示的に `const`指定子を指定して) `const`オブジェクトへのポインタに代入できる。ただし、そのポインタを使用してオブジェクトの値を変更することはできない。次にその例を示す。

```
const int i = 0;
int j = 1;
const int *p = &i; /* Explicit const specifier required */
int *q = &j;
*p = 1;           /* Error -- attempt to modify a const-
                  qualified object through a pointer */
*q = 1;           /* OK */
```

- 非`const`修飾型へのポインタを使用して `const`オブジェクトを修正しようとすると、予測できない結果が生じる。

3.7.2 volatile 型修飾子

`volatile`型修飾子を含むオブジェクトは、そのオブジェクトの参照または修正を変更するコンパイラの最適化の対象としてはならないことを示しています。

注意

`volatile`オブジェクトは、特に副作用を引き起こす傾向があります (第 2.5 節を参照してください)。

`volatile` を使用することによって無効になる最適化は、次のように分類できます。

- オブジェクトの存続期間を変更する最適化
たとえば、オブジェクトの参照がプログラムの別の部分にシフトまたは移動する場合
- オブジェクトのローカル性を変更する最適化

たとえば、ループ・カウンタとして機能している変数をレジスタに格納し、メモリ参照の回数を少なくする場合

- オブジェクトの存在を変更する最適化

たとえば、変数参照を実際に排除するためループを導入する場合

`volatile`指定子を持たないオブジェクトがコンパイラに対してこのような最適化を強制することはありません。コンパイラは、プログラムの状況とコンパイラ最適化レベルによって、最適化を自由に適用します。

`volatile`修飾子は、コンパイラに `volatile`オブジェクトのメモリを割り当てさせて、そのオブジェクトを常にメモリからアクセスさせます。この修飾子は、コンパイラを制御する方法以外の方法でオブジェクトをアクセスできるように宣言する際によく使用されます。したがって、`volatile`キーワードで修飾したオブジェクトは、他のプロセッサまたはハードウェアによってそれぞれの方法で変更またはアクセスされる可能性があります。さらに、副作用の影響を特に受けやすくなります。

次の規則は、`volatile`修飾子を使用する場合に適用されます。

- `volatile`修飾子を使用して、文字列または共用体の単一メンバを含む任意のデータ型を修飾できる。
- `volatile`キーワードを重複して使用すると、警告メッセージが出される。次にその例を示す。

```
volatile volatile int x;
```

- `volatile`を集成体宣言に使用すると、集成体のすべてのメンバが `volatile` で修飾される。集成体の中のあるメンバを修飾するために `volatile` を使用すると、そのメンバのみが修飾される。次にその例を示す。

データ型

3.7 型修飾子

```
volatile struct employee {
    char *name;
    int  birthdate; /* name, birthdate, job_code, and salary are */
    int  job_code;  /* treated as though declared with volatile. */
    float salary;
    } a,b;          /* All members of a and b are volatile-qualified */
struct employee2 {
    char *name;
    volatile int birthdate; /* Only this member is qualified */
    int job_code;
    float salary;
    } c, d;
```

このプログラムの後の部分で別の構造体を指定するために employee タグを使用すると、特に指定がない限り volatile 修飾子は新しい構造体メンバに適用されない。

- const 修飾子は volatile 修飾子と一緒に使用できる。これは、ソース・プロセスでは変わらないが、他のプロセスでは変更可能であるデータ・オブジェクトの宣言に役立つ。または、実時間クロックなどのメモリ・マップ入力ポートのモデルとしても役立つ。
- 非volatile オブジェクトのアドレスは、volatile オブジェクトへのポインタに代入できる。次にその例を示す。

```
const int *intptr;
volatile int x;
intptr = &x;
```

同様に、volatile オブジェクトのアドレスは非volatile オブジェクトへのポインタに代入できる。

3.7.3 __unaligned 型修飾子

ポインタ定義で、指しているデータが正しいアドレスで適切に境界調整されていないことをコンパイラに示すには、このデータ型修飾子を使用します。適切に境界調整されるようにするには、オブジェクトのアドレスはその型のサイズの倍数でなければなりません。たとえば、2 バイトのオブジェクトは偶数のアドレスで境界調整される必要があります。

`__unaligned`と宣言したポインタでデータをアクセスすると、コンパイラはデータをコピーまたは格納するために必要な追加のコードを生成して、境界調整のエラーが出ないようにします。境界調整の誤ったデータは一切使用しないのが良いのですが、パックされた構造体にアクセスする必要やその他の理由から、使用される場合があります。

次は、`__unaligned`の一般的な使用例です。

```
typedef enum {int_kind, float_kind, double_kind} kind;
void foo(void *ptr, kind k) {
    switch (k) {
        case int_kind:
            printf("%d", *(__unaligned int *)ptr);
            break;
        case float_kind:
            printf("%f", *(__unaligned float *)ptr);
            break;
        case double_kind:
            printf("%f", *(__unaligned double *)ptr);
            break;
    }
}
```

3.7.4 `__restrict` 型修飾子

ポインタについてコンパイラによる最適化を行うことを示すには、ポインタ型の宣言で`__restrict`型修飾子を使用します。限定ポインタは、ISO C 規格の 9X リビジョンに追加される予定です。限定ポインタを適切に使用すると、多くの場合、コンパイラによるコード出力の質を改善することができます。

3.7.4.1 理由

以降の各項では、限定ポインタをサポートする理由について説明します。

3.7.4.1.1 エイリアシング

単に値をレジスタに保持しておくことからループの並列実行に至るまで、多くのコンパイラの最適化を行うためには、2つの異なる左辺値が異なるオブジェクトを示しているかどうかを判断する必要があります。オブジェクトが異なっていない場合、それらの左辺値はエイリアスであると言われます。2つの左辺値がエイリアスであるかどうかをコンパイラが判断できない場合は、それらがエイリアスであると仮定して、さまざまな最適化をやめなければなりません。

単一の関数内、あるいは単一のコンパイル単位内でさえ、2つのポインタが同じオブジェクトを指しているかどうかを判断するために利用できる情報は十分でないため、ポインタを介したエイリアシングは大きな困難を伴います。また、情報が十分に利用できたとしても、この分析にはかなりの時間とスペースを必要とします。たとえば、関数の仮引数であるポインタが取り得る値を判断するには、プログラム全体を分析する必要があります。

3.7.4.1.2 ライブラリの例

2つの標準Cライブラリ関数`memmove`と`memcpy`のCの実現において、どのように潜在的なエイリアシングが入るかについて考えてみます。

- `memmove`の使用に関する制限はなく、次に示す実現例では、一時的な配列を経由してコピーすることにより、改訂ISO C規格に記述のあるモデルに準拠している。
- `memcpy`は、相互に重なり合っている配列間でのコピーには使用できないため、実現では直接コピーを行っていることがある。

次の例は、`memcpy`関数と`memmove`関数の実現のサンプルを対比させています。

```
/* Sample implementation of memmove */  
void *memmove(void *s1, const void *s2, size_t n) {  
    char * t1 = s1;  
    const char * t2 = s2;  
    char * t3 = malloc(n);  
    size_t i;  
    for(i=0; i<n; i++) t3[i] = t2[i];  
    for(i=0; i<n; i++) t1[i] = t3[i];  
    free(t3);  
    return s1;  
}
```

```
/* Sample implementation of memcpy */
void *memcpy(void *s1, const void *s2, size_t n);
char * t1 = s1;
const char * t2 = s2;
while(n-- > 0) *t1++ = *t2++;
return s1;
}
```

memcpyの制限は、規格の説明には示されていますが、Cの実現において直接表すことはできません。このため、memmoveでの一時配列の使用を削除するというソース・レベルの最適化はできますが、結果として得られる単一のループに対してのコンパイラの最適化は提供されません。

多くのアーキテクチャでは、バイトをブロック単位でコピーする方が、一度に1バイトづつコピーするよりも速く処理することができます。

- memmoveの実現では、mallocを使用して一時配列を取得することにより、確実に一時配列がソースおよび目的の配列から分離される。このことから、コンパイラは、両方のループに対してブロック・コピーが問題なく使用できることを推断することができる(コンパイラがmallocを新しいメモリを割り当てる特殊な関数として認識する場合)。
- 一方、memcpyの実現では、たとえば、s1とs2が連続するバイトを指している可能性をコンパイラが除外する根拠を提供しない。したがって、ブロック・コピーを無条件に使用することは安全とは思われず、memcpyの単一のループに対して生成されたコードは、memmoveの各ループに対して生成されたコードほど高速ではない可能性がある。

3.7.4.1.3 相互に重なり合うオブジェクト

memcpyの規格での説明にある制限では、相互に重なり合うオブジェクト間でのコピーを禁止しています。オブジェクトとはデータ記憶域の領域のことであり、ビット・フィールドを除いて、1あるいはそれ以上のバイトの連続した並び(数、順番、およびエンコーディングは、明示的に指定されているか、または実装者が定義する。)で構成されます。

次の例について考えてみてください。

```
/* memcpy between rows of a matrix */  
void f1(void) {  
    extern char a[2][N];  
    memcpy(a[1], a[0], N);  
}
```

この例から、次のことがわかります。

- オブジェクトは、厳密に、ポインタで指し示されているデータ記憶域の領域であり、長さは動的にNバイトに決定される。つまり、N個の要素を持つ文字型の配列として処理される。
- オブジェクトは、実引数が指していると解釈できる最大のオブジェクトではない。
- memcpyの呼出しは、定義どおりに動作する。
- ポインタがそれぞれ異なる (相互に重なり合っていない) オブジェクトを指しているので、動作は未定義にならない。

では、次の例について考えてみましょう。

```
/* memcpy between halves of an array */  
void f2(void) {  
    extern char b[2*N];  
    memcpy(b+N, b, N);  
}
```

この例からは、次のことがわかります。

- オブジェクトは、宣言や型に関係のないデータ記憶域の領域として定義される。
- memcpyでは、配列内の連続する一連の要素は、それ自身でオブジェクトと見なされることができる。
- オブジェクトは、解釈できる最小の連続する一連のバイトではなく、厳密に、ポインタから始まり N バイトの長さのデータ記憶域の領域である。
- 配列bの重なり合っていない半分は、それ自身でオブジェクトと見なされる。

- 定義どおりの動作になる。

オブジェクトの長さは、さまざまな方法で決定されます。

- すべての要素がアクセスされる文字列では、長さはヌル・バイトの終了で推断される。
- `mbstowcs`, `wcstombs`, `strftime`, `vsprintf`, `sscanf`, `sprintf`, および他のすべての同様の関数では、オブジェクトと長さは動的に決定される。

3.7.4.1.4 `memcpy` のための限定ポインタのプロトタイプ

`memcpy`に関する制限と同様、エイリアシング制限も関数定義として表すことができるのであれば、コンパイラは効果的にポインタのエイリアスの分析を行うために利用することができますでしょう。`__restrict`型修飾子は、ポインタの宣言で、ポインタが`malloc`への呼出しで初期化されたかのように、ポインタが指し示すオブジェクトに対して、排他的な初期のアクセスを提供することを指定することによって、これを実現します。

次に示す`memcpy`のためのプロトタイプは、必要な制限を表すとともに、現在のプロトタイプと互換性があります。

```
void *memcpy(void * __restrict s1, const void * __restrict s2, size_t n);
```

3.7.4.2 `__restrict` 型修飾子の正式な定義

次に示す限定ポインタの定義では、できるだけ多数の典型的な例におけるエイリアシング制限の式をサポートしています。これは、既存のプログラムを変換して、限定ポインタを使用する際に有用であり、また、新規のプログラムでは、より自由にスタイルを選択できるようになります。

このため、この定義では、限定ポインタについて次のことが可能です。

- 修正可能
- 構造体のメンバと配列の要素
- ネストされているブロック内で宣言された限定ポインタは、そのブロック内でのみ非エイリアシング表明をする、という意味で強力にスコープを持つ

定義

ポインタは、宣言で `__restrict` 型修飾子を指定することにより、限定ポインタとして指定されます。

改訂 ISO C 規格に含まれる予定の限定ポインタの正式な定義は、次のとおりです。

D は、オブジェクト P を限定修飾ポインタとして指定する方法を提供する通常の識別子の宣言とする。

D がブロックの内部にあり、記憶域クラス `extern` を持たないとき、B はそのブロックを示すものとする。D が関数定義の仮引数宣言の並びにあるとき、B はそれに関連するブロックを示すものとする。それ以外の場合には、B は `main` のブロック (または、フリースタANDING 環境において、プログラムの開始時に呼ばれる関数のブロック) を示す。

以降で、(ポインタ式 E を評価する前であって B を実行しているある評価順序点において) P が以前指していた配列オブジェクトのコピーを指すように P を修正すると E の値が変わるとき、E はオブジェクト P に基づいているという。つまり、E は、P を介して間接的に参照されるオブジェクトの値ではなく、P の値自体に依存している。たとえば、識別子 `p` が `(int ** restrict)` という型を持つとすると、ポインタ式 `p` および `p+1` は、`p` で指定された限定ポインタ・オブジェクトに基づいているが、ポインタ式 `*p` および `p[1]` は基づいていない。

B の各実行中には、O は、P に基づくポインタ式を介するすべての参照によって動的に決定される配列オブジェクトとする。O の値へのすべての参照は、P に基づくポインタ式を介して行われなければならない。さらに、P が、別の限定ポインタ・オブジェクト P2 に基づいているポインタ式 E の値を割り当てられ、ブロック B2 に関連付けられていれば、B2 のいずれかの実行は、B の実行の前に開始されるか、あるいは、B2 の実行はその割り当ての前に終了しなければならない。この条件を満足しない場合、動作は未定義となる。

ここで、Bの実行とは、Bに関連付けられていて自動記憶域存続期間を持つオブジェクトのインスタンスのために記憶域の予約が保証されている間の、プログラムの実行の一部という意味である。値の参照とは、値へのアクセスまたは値の変更のことである。Bの実行中、実際に評価される参照だけが調べられる。(評価されない式での参照や、可視識別子を使用するという意味において「利用可能」であるが、実際にはBのテキストに現れない参照は除外される。)

翻訳プログラムは、限定ポインタの使用によるすべてまたは任意のエイリアシングを自由に無視することができる。

3.7.4.3 例

`__restrict`型修飾子の正式な定義は分かりにくいものですが、説明を簡略にすると不正確で不完全になってしまいます。この定義の本質は、`__restrict`型修飾子は、限定ポインタを介してメモリ・アクセスが行われるときにはいつでもプログラマによってなされる表明であり、コンパイラが考慮する必要のある唯一のエイリアスは、同じポインタを介して行われる別のアクセスである、ということです。

複雑に感じられる原因の多くは、ポインタを介して行われるアクセスが何を意味するのかを厳密に定義したり(基づくの規則)、ブロック境界においてのみ起こりうるエイリアシングを制限しながら、限定ポインタに別の限定ポインタの値を代入する方法を指定したりする点にあります。限定ポインタについて理解する最善の方法は、例を参照することです。

次の例は、さまざまな文脈における限定ポインタの使用方を示しています。

3.7.4.3.1 ファイル・スコープ限定ポインタ

ファイル・スコープ限定ポインタは、非常に強い制限を受けます。このポインタは、プログラムが存続している間、単一の配列オブジェクトを指していなければなりません。その配列オブジェクトは、限定ポインタを介して参照されてもならず、その宣言された名前(ある場合)または別の限定ポインタを介して参照されてもなりません。

これらの制限のため、ポインタを介した参照は、その宣言された名前で静的配列を参照するのと同様に効率的に最適化することができます。したがって、ファイル・スコープ限定ポインタは、動的に割り当てられるグローバル配列へのアクセスを提供する際に有用です。

次の例では、コンパイラは__restrict型修飾子から、名前a, b, およびcの間で潜在的なエイリアシングが存在しないことを推測することができます。

```
/* File Scope Restricted Pointer */  
  
float * __restrict a, * __restrict b;  
float c[100];  
  
int init(int n) {  
    float * t = malloc(2*n*sizeof(float));  
    a = t;      /* a refers to 1st half. */  
    b = t + n;  /* b refers to 2nd half. */  
}
```

関数initでは、割り当てられた記憶域の単一のブロックが2つの別個の配列に分けられています。

3.7.4.3.2 関数の仮引数

限定ポインタは関数のポインタ仮引数としても非常に有用です。次の例について考慮してみてください。

```
/* Restricted pointer function parameters */  
  
float x[100];  
float *c;  
  
void f3(int n, float * __restrict a, float * const b) {  
    int i;  
    for ( i=0; i<n; i++ )  
        a[i] = b[i] + c[i];  
}  
  
void g3(void) {  
    float d[100], e[100];  
    c = x; f3(100, d, e); /* Behavior defined. */  
           f3( 50, d, d+50); /* Behavior defined. */  
           f3( 99, d+1, d); /* Behavior undefined. */  
    c = d; f3( 99, d+1, e); /* Behavior undefined. */  
           f3( 99, e, d+1); /* Behavior defined. */  
}
```

関数f3では、コンパイラは、変更されたオブジェクトのエイリアシングはないと推断して、積極的にループの最適化を行う可能性があります。f3に入ると、限定ポインタaはそれに関連する配列への排他的なアクセスを提供する必要があります。

す。特に、f3の内部では、bもcもaに基づくポインタ値を代入されていないため、どちらもaに関連する配列内を指していない可能性があります。bについては、宣言のconst修飾子から明らかですが、cについては、f3の本体の検査が必要になります。

g3に示す2つの呼出しは、結果として、__restrict修飾子と矛盾するエイリアシングとなるため、それらの動作は未定義となります。cが、bに関連している配列内を指すことは許されています。また、この目的のため、特定のポインタに関連する「配列」は、そのポインタを介して実際に参照される配列オブジェクトの一部分のみを意味します。

3.7.4.3.3 ブロック・スコープ

ブロック・スコープ限定ポインタは、そのブロックに制限されたエイリアシング表明を行います。これは、表明に関数スコープを持たせるよりも自然です。たとえば、主要なループにのみ適用されるローカル表明ができるようにします。また、関数をマクロに変換することにより関数をインライン化する際に同様の表明を行うことができるようにします。

次の例では、元の限定ポインタ仮引数は、ブロック・スコープ限定ポインタで表されています。

```
/* Macro version of f3 */
float x[100];
float *c;

#define f3(N, A, B) \
{ \
    int n = (N); \
    float * __restrict a = (A); \
    float * const b = (B); \
    int i; \
    for ( i=0; i<n; i++ ) \
        a[i] = b[i] + c[i]; \
}
```

3.7.4.3.4 構造体のメンバ

構造体の限定ポインタ・メンバはエイリアシング表明を行います。この表明のスコープは、この構造体にアクセスするために使用される通常の識別子のスコープです。

したがって、次の例では、構造体の型はファイル・スコープで宣言されていますが、f4の仮引数の宣言によって行われている表明は、(関数の) ブロック・スコープを持ちます。

```
/* Restricted pointers as members of a structure */
struct t {      /* Restricted pointers assert that      */
    int n;      /* members point to disjoint storage. */
    float * __restrict p;
    float * __restrict q;
};

void f4(struct t r, struct t s) {
    /* r.p, r.q, s.p, s.q should all point to      */
    /* disjoint storage during each execution of f4. */
    /* ... */
}
```

3.7.4.3.5 型定義

typedef内の__restrict修飾子は、オブジェクトへのアクセスを提供する通常の識別子の宣言でtypedef名を使用する際にエイリアシング表明を行います。構造体のメンバに関しては、typedef名のスコープではなく、通常の識別子のスコープが、エイリアシング表明のスコープを決定します。

3.7.4.3.6 限定ポインタに基づく式

次の例について考慮してみてください。

```
/* Pointer expressions based on p */
#include <stdlib.h>
#include <string.h>
struct t { int * q; int i; } a[2] = { /* ... */ };
```

```
void f5(struct t * __restrict p, int c)
{
    struct t * q;
    int n;
    if(c) {
        struct t * r;
        r = malloc(2*sizeof(*p));
        memcpy(r, p, 2*sizeof(*p));
        p = r;
    }
    q = p;
    n = (int)p;

    /* -----
       Pointer expressions      Pointer expressions
       based on p:              not based on p:
       -----
       p                        p->q
       p+1                      p[1].q
       &p[1]                    &p
       &p[1].i
       q                        q->p
       ++q                      (char *) (p->i)
       (char *)p                ((struct t *)n)->q
       (struct t *)n
       ----- */
}

main() {
    f5(a, 0);
    f5(a, 1);
}
```

この例では、限定ポインタ仮引数`p`は、2つの構造体の元の配列のコピーを指すように潜在的に調整されます。定義により、その後のポインタ式は、その値がこの調整によって変更される場合に限り、`p`に基づくと言われます。

コメント部分の説明は次のとおりです。

- 最初の欄のポインタ式の値はこの調整によって変更されるため、これらの式は`p`に基づいている。

- 2 番目の欄のポインタ式の値はこの調整によって変更されないため、これらの式はpに基づいていない。

この式に関する適切な print 文を追加し、main内でのf5の2つの呼出しによって生成された値を比較することにより、このことを確認することができます。

「基づく」の定義は、処理系定義の動作に依存する式に適用されます。これについては例を参照してください。例では、後ろに(struct t *)が続く(int)のキャストは、元の値を提供すると想定しています。

3.7.4.3.7 限定ポインタ間での代入

最初の限定ポインタが関連付けられているブロックの実行を、2 番目のポインタに関連付けられているブロックの後に開始した場合、一方の限定ポインタがもう一方の限定ポインタより「新しい」と想定します。このとき、正式な定義では、新しい限定ポインタには、古い限定ポインタに基づく値を代入することができます。これにより、たとえば、限定ポインタ仮引数を持つ関数を、限定ポインタである実引数を指定して呼び出すことができます。

逆に、古い限定ポインタには、新しい限定ポインタに関連付けられているブロックの実行が終了した後にのみ、新しい限定ポインタに基づく値を代入することができます。これにより、たとえば、関数にローカルな限定ポインタの値を関数が返した後、その返却値を別の限定ポインタに代入することができるようになります。

プログラムに、これら2つのカテゴリのいずれにも当てはまらない2つの限定ポインタ間での代入が含まれている場合、このプログラムの動作は未定義となります。次の例を参照してください。

```
/* Assignments between restricted pointers */  
int * __restrict p1, * __restrict p2;
```

```
void f6(int * __restrict q1, * __restrict q2)
{
    q1 = p1;    /* Valid behavior    */
    p1 = p2;    /* Behavior undefined */
    p1 = q1;    /* Behavior undefined */
    q1 = q2;    /* Behavior undefined */
    {
        int * __restrict r1, * __restrict r2;
        ...
        r1 = p1; /* Valid behavior    */
        r1 = q1; /* Valid behavior    */
        r1 = r2; /* Behavior undefined */
        q1 = r1; /* Behavior undefined */
        p1 = r1; /* Behavior undefined */
        ...
    }
}
```

3.7.4.3.8 非限定ポインタへの代入

次の例のように、限定ポインタの値は、非限定ポインタに代入することができます。

```
/* Assignments to unrestricted pointers */
void f7(int n, float * __restrict r, float * __restrict s) {
    float * p = r, * q = s;
    while(n-- > 0)
        *p++ = *q++;
}
```

*Compaq C*コンパイラは、ポインタ値を追跡して、限定ポインタ`r`と`s`が直接使用された場合と同様に、効率的にループを最適化します。これは、この場合、`p`は`r`に基づき、`q`は`s`に基づいていると簡単に判断できるためです。

限定ポインタと非限定ポインタをより複雑に組み合わせて使用すると、難しすぎてコンパイラが分析できないため、あまり効率的ではありません。パフォーマンスが重要な場合には、プログラミングのスタイルをコンパイラの機能に合わせる必要があります。保守的な方法では、同じ関数内では、限定ポインタと非限定ポインタの両方を使用しないようにします。

3.7.4.3.9 型修飾子の非効率的な使用

正式な定義で特別に記述している場合を除いて，`__restrict`修飾子は`const`および`volatile`と同様に動作します。

特に，関数の返却値の型またはキャストの型名を修飾することは制約違反ではありませんが，関数呼出し式およびキャスト式は左辺値ではないため，修飾子は何の影響も及ぼしません。

このため，次の例で`f8`の宣言に`__restrict`修飾子が指定されていても，`f8`を呼び出す関数内のエイリアシングについては何の表明も行いません。

```
/* Qualified function return type and casts */
float * __restrict f8(void) /* No assertion about aliasing. */
{
    extern int i, *p, *q, *r;
    r = (int * __restrict)q; /* No assertion about aliasing. */
    for(i=0; i<100; i++)
        *(int * __restrict)p++ =  r[i]; /* No assertion      */
                                         /* about aliasing. */
    return p;
}
```

同様に，2つのキャストは，ポインタ`p`および`r`を介した参照のエイリアシングについて，何の表明も行いません。

3.7.4.3.10 制約違反

ポインタ型でないオブジェクト型を限定修飾することや，関数へのポインタを限定修飾することは，制約違反です。次の例を参照してください。

```
/* __restrict cannot qualify non-pointer object types: */
int __restrict x; /* Constraint violation */
int __restrict *p; /* Constraint violation */

/* __restrict cannot qualify pointers to functions: */
float (* __restrict f9)(void); /* Constraint violation */
```

3.8 型定義

`typedef` は型の同義語を定義するために使用します。この定義では、識別子はオブジェクトの代わりに型の名前を指定します。この定義方法によって、長い型定義や間違いやすい型定義に省略名を定義できます。

型定義では新しい基本データ型を作成しません。これは、基本型または派生型の別名を作成します。たとえば、次のコードを使用すると、プログラムの後の部分で使用するオブジェクトのデータ型を説明するときに便利です。

```
typedef float *floatp, (*float_func_p)();
```

この場合、`floatp`型は「float値へのポインタ」型であり、`float_func_p`型は「float を返す関数へのポインタ」型です。

型定義は、完全な型名が通常使用されている (通常の型名を使用できる) 場所であれば、どこでも使用できます。型定義は変数と同じ名前空間を共有し、定義した型はそれに等しい型と完全に互換性があります。修飾型として定義された型はその型修飾を引き継ぎます。

他の型定義から型定義を作成することもできます。次にその例を示します。

```
typedef char byte;
typedef byte ten_bytes[10];
```

型定義は変数または関数に適用できます。型定義を他の指定子と合わせて使用することはできません。次にその例を示します。

```
typedef int *int_p;
typedef unsigned int *uint_p;
unsigned int_p x;          /* Invalid */
uint_p y;                  /* Valid  */
```

型定義を使用して、関数型を宣言することもできます。ただし、型定義は関数定義に使用することはできません。関数の返却値の型は型定義を使用して指定できます。次にその例を示します。

データ型

3.8 型定義

```
typedef unsigned *uint_p; /* uint_p has type "pointer to unsigned int" */
uint_p xp;
typedef uint_p func(void); /* func has type "function returning pointer to */
                          /* unsigned int */
func f;
func b;
    func f(void)          /* Invalid -- this declaration specifies a */
                          /* function returning a function type, which */
    {                    /* is not allowed */
        return xp;
    }
uint_p b(void)            /* Legal -- this function returns a value of */
    {                    /* type uint_p. */
        return xp;
    }
```

次の例では、関数定義は typedef 名から引き継ぐことができないことを示しています。

```
typedef int func(int x);
func f;
func f /* Valid definition of f with type func */
{
    return 3;
} /* Invalid, because the function's type is not inherited */
```

この例を有効な形式に変更すると、次のようになります。

```
typedef int func(int x);
func f;
int f(int x) /* Valid definition of f with type func */
{
    return 3;
} /* Legal, because the function's type is specified */
```

typedef 名に仮引数名を含めて、プロトタイプ情報を取り込むことができます。また、第 2.3 節で説明したスコープ規則に従って、内部スコープに typedef 名を再定義することもできます。

宣言はプログラムに使用する識別子を導入し，その識別子の重要な属性 (型，記憶域クラス，および識別子名など) を指定するために使用します。また，オブジェクト用に記憶域を確保したり，または関数本体を含む宣言を定義と呼びます。

一般的な宣言の構文規則については第 4.1 節を，初期化については第 4.2 節を，そして外部宣言については第 4.3 節を参照してください。

次の種類の識別子を宣言することができます。各宣言と初期化の構文については，関連する節を参照してください。関数については第 5 章で説明します。

- 単純オブジェクト (第 4.4 節)
- 列挙定数 (第 4.5 節)
- ポインタ (第 4.6 節)
- 配列 (第 4.7 節)
- 構造体メンバと共用体メンバ (第 4.8 節)
- タグ (第 4.10 節)

注意

`#define`命令で作成した前処理マクロは宣言ではありません。前処理命令でマクロを作成する方法については，第 8 章を参照してください。

4.1 宣言の構文規則

宣言の一般的な構文は、次のとおりです。

宣言:

宣言指定子 初期宣言子並び_{opt} ;

宣言指定子:

記憶域クラス指定子 宣言指定子_{opt}

型指定子 宣言指定子_{opt}

型修飾子 宣言指定子_{opt}

初期宣言子並び:

初期宣言子

初期宣言子並び , 初期宣言子

初期宣言子:

宣言子

宣言子 = 初期化子

次は、宣言の一般的な構文に関する注意事項です。

- 記憶域クラス指定子、型修飾子、および型指定子は任意の順序で指定できます。これらはすべてオプションですが、関数宣言以外では指定子または修飾子が最低 1 つは必要です。記憶域クラス指定子を宣言の始まり以外に置くのは、旧形式の宣言です。
- 記憶域クラス・キーワードは `auto` , `static` , `extern` , および `register` です。
- 型修飾子は `const` および `volatile` です。
- 宣言子は、宣言するオブジェクトまたは関数の名前です。宣言子は単一の識別子のように単純にすることもできる他に、配列、構造体、ポインタ、共用体、または関数 (`*x` , `tree()` , および `treebar[10]` など) を宣言する複雑な構成にすることもできます。

完全宣言子は、他の宣言子の一部ではない宣言子です。完全宣言子の末尾はシーケンス・ポイントです。完全宣言子の中でネストした宣言子のシーケンスに可変長配列型が含まれている場合、完全宣言子によって指定されている型は可変修飾されているといえます。

- 初期化子はオプションであり、オブジェクトの初期値を与えます。初期化子は宣言するオブジェクトの型によって、単一の値または中括弧で囲んだ一連の値にすることができます。
- 宣言は、識別子のスコープの始まりを決定します。
- 識別子の結合は、宣言の位置と指定した記憶域クラスによって決定します。

次にその例を示します。

```
volatile static int data = 10;
```

この宣言は、修飾型 (型修飾子を持つデータ型。この例では `volatile` が `int` を修飾している。)、記憶域クラス (`static`)、宣言子 (`data`)、および初期値 (`10`) を示しています。また、記憶域がデータ・オブジェクト `data` 用に確保されるため、この宣言は定義であるともいえます。

これは単純な例ですが、複雑な宣言の場合は解釈するのがより難しくなります。*Compaq C* 宣言の解釈については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

次は宣言に適用される意味規則です。

- 空宣言は許可されていない。宣言には最低 1 つの宣言子、構造体タグ、共用体タグ、または列挙型メンバを指定しなければならない。
- 各宣言子は 1 つの識別子を宣言する。宣言の宣言子数に制限はない。
- 各オブジェクトの宣言には、最低 1 つの記憶域クラス指定子を使用できる。何も指定されない場合は、関数定義内に宣言されるオブジェクトには `auto` 記憶域クラスが割り付けられ、関数定義外で宣言されるオブジェクトには `extern` 記憶域クラスが割り付けられる。
- ブロック・スコープを持つ関数宣言に使用可能な唯一の (オプションの) 記憶域クラスは `extern`。

宣言

4.1 宣言の構文規則

- 型指定子がない場合には、省略時の設定は `signed int` になる。
- 宣言子は、宣言子のスコープが決定するプログラムの一定の範囲内でのみ使用できる。宣言子の記憶域割付けの存続期間は、その記憶域クラスによって異なる。スコープについては第 2.3 節を、記憶域クラスについては第 2.10 節を参照すること。
- 識別子の有効性はその可視性によって制限でき、プログラム内で部分的に隠すことができる。可視性については第 2.4 節を参照すること。
- 同じオブジェクトまたは関数を参照する同じスコープの宣言は、すべて互換型を持たなければならない。
- オブジェクトに結合がない場合、同じ名前空間にあって同じスコープを持つオブジェクトの宣言は 1 つしか存在できない。結合を持たないオブジェクトは、宣言の終わりまたは最後の初期化子 (存在する場合) によって完了しなければならない。結合については、第 2.8 節を参照すること。

記憶域割付け

記憶域は、次の環境でデータ・オブジェクトに割り付けられます。

- オブジェクトに結合がない場合には、オブジェクトの宣言時に記憶域が割り付けられます。`auto`記憶域クラスまたは `register`記憶域クラスを持つブロック・スコープ・オブジェクトが宣言されると、ブロックの終わりでその記憶域割付けが解除されます。
- オブジェクトが内部結合を持つ場合には、オブジェクトの最初の定義時に記憶域が割り付けられます。
- オブジェクトが外部結合を持つ場合には、オブジェクトの初期化時に記憶域が割り付けられます。これは、各オブジェクトに対して 1 回だけ行われます。オブジェクトが仮定義 (第 2.9 節を参照) のみを持つ場合には、コンパイラは 0 の初期化子を持つオブジェクトのファイル・スコープ定義があるものとみなします。結合についての詳細は、第 2.8 節を参照してください。

注意

コンパイラは、必ずしもソース・コードでの宣言順序に従って、個別の変数をメモリ位置に割り当てるとはかぎりません。さらに、ソース・コード、コマンド行オプションに対する表面的には関係のない変更、あるいはコンパイラのバージョンが 1 つ新しくなったことにより、割り当ての順序が変わる可能性があります。これは、本質的に予測不可能です。変数の相

互の位置を制御するための唯一の方法は、それらの変数を同一struct型のメンバとすることです。または、OpenVMS Alphaシステムでは、`#pragma extern_model strict_refdef`でnoreorder属性を使用します。

4.2 初期化

初期化子は、オブジェクトに初期値を提供します。構文は次のとおりです。

初期化子:

```
代入式  
{ 初期化子並び }  
{ 初期化子並び , }
```

初期化子並び:

```
ディジグネーションopt初期化子  
初期化子並び , ディジグネーションopt初期化子
```

ディジグネーション:

```
指名子並び=
```

指名子並び:

```
指名子  
指名子並び指名子
```

指名子:

```
[ 定数式 ]  
. 識別子
```

各型のオブジェクトの初期化については、これ以降の各節で説明します。ただし、次の規則は`Compaq C`におけるすべての初期化に適用されます。

- 初期化子の数は、初期化するオブジェクトの数を超えてはならない。初期化子には初期化するオブジェクトの数よりも少ない数を指定できるが、この場合、残りのオブジェクトは0に初期化される。

宣言

4.2 初期化

- 定数式は静的記憶域存続期間を持つオブジェクトの初期値に使用するか、集合体型または共用体型を持つオブジェクトの初期値並びに使用しなければならない。
- 識別子の宣言がブロック・スコープを持ち、さらにその識別子が外部結合または内部結合を持つ場合には、その識別子の宣言に初期化子を含めることはできない。
- 静的記憶域存続期間を持つオブジェクトが明示的に初期化されない場合には、数値型を持つ各メンバには0が代入され、ポインタ型を持つ各メンバには空ポインタ定数が代入されて、暗黙に初期化される。自動記憶域存続期間を持つオブジェクトが明示的に初期化されない場合には、その値は不定である。
- スカラ・オブジェクトの初期化子は単式でなければならず、中括弧で囲むこともできる。オブジェクトの初期値は式の初期値。単純代入と同じ種類の規則と変換が適用される。
- 集合体オブジェクトが集合体または共用体であるメンバを含む場合、あるいは共用体の最初のメンバが集合体または共用体である場合には、その初期化の規則は集合体メンバまたは含まれている共用体に再帰的に適用される。初期化子並びが集合体メンバまたは含まれている共用体で使用されると、その並び内の初期化子が集合体メンバまたは含まれている共用体のメンバを初期化する。また、並び内の初期化子がオブジェクトに使用される数よりも多い場合には、その並びに残っている任意のメンバは、集合体オブジェクトの次のメンバを初期化するために残される。次にその例を示す。

```
struct t1 {
    int i;
    double d;
};

union t2 {
    int i;
    double d;
};

struct t3 {
    struct t1 s;
    union t2 u;
};
```



```
struct t3 st[] = { /* complete initializer */
    1, 2, 0, 4, 0, 0, 7, 0, 0
};
```

この宣言の場合、st変数は3つの構造体の配列である。その初期の内容は次のとおりである。

	s	u
	-----	-
st[0]:	1, 2.0,	0
st[1]:	4, 0.0,	0
st[2]:	7, 0.0,	0

また、この変数は次の方法でも定義できる。4つの初期値はすべて等しくなる。

```
struct t3 st[] = { /* partial initializer */
    1, 2, 0, 4, 0, 0, 7
};

struct t3 st[] = { /* nested and complete initializers */
    {1, 2, 0},
    {4, 0, 0},
    {7, 0, 0}
};

struct t3 st[] = { /* nested and partial initializers */
    {1, 2},
    {4},
    {7}
};
```

配列、構造体、共用体の初期化については、4.7.1、4.8.4、4.8.5を参照すること。

- 配列および構造体にディジグネーションを使用した初期化子については、第4.9節を参照すること。
- 変形構造体と変形共用体は、通常の構造体と共用体と同様に初期化される。詳細については、第4.8.4項と第4.8.5項を参照すること。

*Compaq C*では、慣習として初期化子を余分な中括弧で囲むことを許可しています。たとえば、書式をわかりやすくするために中括弧で囲みます。このような初期化子は、使用するパーサの型によって解析が異なります。*Compaq C*ではANSI規格によって指定されている解析方法を使用しており、これはトップダウン解析として知られています。部分的に中括弧で囲んだ初期化子をボトムアップ解析を採用したプログラムで使用した場合、その結果は保証できません。コモン C 互換性モードで不要な中括弧を検出した場合や、コマンド行でエラーを検査するコンパイル・オプションが指定された場合には、コンパイラは警告メッセージを出します。

4.3 外部宣言

関数の外にあるオブジェクト宣言を外部宣言と呼びます。外部宣言は内部宣言とは対照的なものです。内部宣言は関数またはブロック内で作成され、その関数またはブロックの内部に存在し、その内部でのみ参照できます。コンパイラは内部宣言された識別子を宣言した場所からそのブロックの終わりまで認識します。

オブジェクトの宣言にファイル・スコープと初期値がある場合には、その宣言はオブジェクトの外部宣言でもあります。C 言語プログラムは、オブジェクトと関数の一連の外部定義から構成されています。

定義は、宣言する要素用に記憶域を確保します。次にその例を示します。

```
float fvalue = 15.0;           /* external definition */
main ()
{
    int ivalue = 15;           /* internal definition */
}
```

外部データ宣言と外部関数定義は、データ宣言または関数宣言と同じ形式で指定します (標準関数宣言の構文については第 5 章を参照してください)。適用される規則は次のとおりです。

- 外部的に宣言したオブジェクトの記憶域クラスは未指定のままにするか、あるいは `extern` または `static` として宣言できる (第 2.10 節を参照のこと)。未指定の場合には、省略時の設定は `extern` 記憶域クラスであり、宣言したオブジェクトの結合は外部結合である。また、型指定子はオプションであり、この場合

の省略時の設定は `int` 型である。記憶域クラス指定子、型修飾子、および型指定子のすべてを宣言から省略することはできないので注意すること。

- 外部結合を持つオブジェクトを宣言するか、または式で使用する場合には、プログラム中の別の場所に識別子の外部定義を 1 回だけ指定する必要がある。同じオブジェクトが 2 回以上外部宣言されている場合には、それぞれの宣言の型と結合が一致していなければならない(第 2.8 節を参照のこと)。
- 1 つまたは複数の宣言が不完全にオブジェクト型を指定しており、さらに完了型を持つオブジェクトの宣言が 1 つだけある場合には、すべての宣言がその完了型と一致するように指定される。
- 外部宣言のスコープは、外部宣言が宣言されているファイルの終わりまで有効である。これに対して、内部宣言はそれが宣言されたブロックの終わりまでしか有効ではない。1 つのブロック内のみで使用するデータ・オブジェクトは、そのブロック内で宣言しなければならない。外部定義の構文は、すべての定義の構文と同様である。関数定義は外部レベルでのみ行うことができる。
- `auto` オブジェクトと `register` オブジェクトの外部宣言は、許可されていない。内部宣言した `auto` オブジェクトと `register` オブジェクトは、自動的に初期化されない。明示的に初期化されない場合には、そのアドレスには以前に格納された不適切な値が入る。`static` オブジェクトは、明示的に初期化しない限りすべて自動的に 0 に初期化される。

注意

*Compaq C*では事前に宣言しておかなくても外部関数を呼び出すことはできますが、推奨できる方法ではありません。これでは型検査が行われなため、バグが発生しやすくなります。このような関数呼出しが行われると、コンパイラは `int` 型の外部宣言がその呼出しを含むブロックに指定されている場合と同様に関数を処理します。次にその例を示します。

```
void function1()
{
  int a,b;
  x (a,b)
}
```

この場合、コンパイラは `extern int x();` 宣言が `function1` 定義ブロック内に指定されているかのように処理します。

コンパイル単位の識別子の最初の宣言では明示的に、または `static` キーワードを省略することでそれが内部識別子か外部識別子かを指定しなければなりません。オブジェクトごとに1つずつしか定義を使用できません。同じオブジェクトに対して矛盾する定義または重複する定義を行わない限り、同じオブジェクトに複数の宣言を行うことができます。

外部オブジェクトは、明示的な初期化または仮定義により定義することが可能です。ファイル・スコープを持ち、初期化子を持たず、`static` 以外の記憶域クラス指定子を持つオブジェクトの宣言は仮定義です。オブジェクトの完全定義が見つからない場合には、コンパイラは仮定義をオブジェクトの唯一の定義とみなします。すべての宣言についてオブジェクトが定義されるまでは、実際に記憶域が割り付けられることはありません。

コンパイル単位内の1つのオブジェクトに複数の仮定義があり、そのオブジェクトの外部定義がない場合には、コンパイラは0の初期化子を持つオブジェクト(コンパイル単位の終わりまで合成型を持つオブジェクト)のファイル・スコープ宣言があるものとみなします。合成型の定義については、第2.7節を参照してください。

オブジェクトの宣言が仮定義であり内部結合を持つ場合に宣言される型は、不完全型であってはなりません。仮定義の例については、第2.9節を参照してください。

4.4 単純オブジェクトの宣言

単純オブジェクトとは、基底データ型の 1 つを持つオブジェクトです。したがって、単純オブジェクトは汎整数型または浮動小数点型を持つことができます。すべてのオブジェクトと同様に、単純オブジェクトは名前の付けられた記憶位置であり、その値はプログラムの実行中に変更できます。プログラムで使用する単純オブジェクトは、すべて宣言する必要があります。

単純オブジェクト宣言は以下の要素から構成されます。

- オプションのデータ型指定子キーワード
- オプションの型修飾子キーワード (const または volatile)

次にその例を示す。

```
const int *p;    /* const qualifies the integer p points to */
int *const p;    /* const qualifies the pointer p */
```

- オプションの記憶域クラス・キーワード

記憶域クラス・キーワードが省略されると、プログラム中の宣言の位置によって異なる記憶域クラスが省略時の設定になる。記憶域クラス・キーワードとデータ型キーワードの位置は交換できるが、宣言の開始場所以外に記憶域クラス・キーワードを指定するのは旧形式である。

- 宣言されたオブジェクトの名前を表示する宣言子
- 単純オブジェクトの初期値を与える初期化子

単純オブジェクトの初期化子は等号 (=) とその後に続く単式から構成される。

4.4.1 単純オブジェクトの初期化

単純オブジェクトの初期化子は、等号 (=) とその後に続く単式から構成されます。次にその例を示します。

```
int x = 10;
float y = ((12 - 2) + 25);
```

宣言

4.4 単純オブジェクトの宣言

この場合、宣言は `x` オブジェクトを整数値として宣言し、最初は 10 に等しいものとして定義しています。また、`y` を浮動小数点値として宣言し、初期値 35 を持つものとして定義しています。

初期化子がない場合の `auto` オブジェクトの初期値は定義されていません。明示的な初期化が行われない `static` オブジェクトは、自動的に 0 に初期化されます。オブジェクトが `static` 配列または構造体である場合には、すべてのメンバが 0 に初期化されます。

外部結合または内部結合を持つブロック・スコープの識別子 (`extern` または `static` キーワードを使用して宣言されている識別子) には、その宣言に初期化子を含めることはできません。これは、その識別子が別の場所で初期化されるからです。

4.4.2 整数オブジェクトの宣言

整数オブジェクトは `int`、`long`、`short`、`signed`、および `unsigned` の各キーワードで宣言できます。`char` も使用できますが、小さい値にのみ使用できます。次に、整数宣言の文の例を示します。

```
int x;          /* Declares an integer variable x    */
int y = 10;     /* Declares an integer variable y    */
               /* and sets y's initial value to 10 */
```

使用可能な値の範囲を示すために、いくつかのキーワードを一緒に使用できます。次にその例を示します。

```
unsigned long int a;
signed long;      /* Synonymous with "signed long int" */
unsigned int;
```

オブジェクトに汎整数データ型を選択する場合には、整数オブジェクトを表現できるようにその値の範囲を考慮しなければなりません。汎整数データ型のサイズと範囲については、第 3 章を参照してください。

4.4.3 文字変数の宣言

文字オブジェクトは、charキーワードで宣言します。次の例は、文字オブジェクトの初期化を行う文字宣言を示しています。

```
char ch = 'a'; /* Declares an object ch with an initial value 'a' */
```

*Compaq C*では、文字列リテラルは char型の配列に格納されます。配列についての詳細は、第 4.7 節を参照してください。

4.4.4 浮動小数点変数の宣言

浮動小数点オブジェクトを宣言する場合は、格納されるオブジェクトに必要な精度を決定する必要があります。単精度または倍精度オブジェクトを使用できます。単精度の場合は floatキーワードを使用してください。倍精度の場合は double または long doubleキーワードを使用してください。次にその例を示します。

```
float x = 7.5;  
double y = 3.141596;
```

浮動小数点型の範囲および精度については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

4.5 列挙型の宣言

列挙型はユーザ定義の整数型であり、列挙定数を定義するものです。この列挙定数とは、整数として表現できる値を持つ整数定数式のことです。列挙型の宣言の構文は次のとおりです。

列挙型指定子:

```
enum 識別子opt { 列挙子並び }  
enum 識別子opt { 列挙子並び, }  
enum 識別子
```

宣言

4.5 列挙型の宣言

列挙子並び:

列挙子
列挙子並び , 列挙子

列挙子:

列挙定数
列挙定数 = 定数式

*Compaq C*では, enum型のオブジェクトは互換性があります。

次に, 列挙型と列挙型タグの宣言例を示します。

```
enum shades
{
    off, verydim, dim, prettybright, bright
} light;
```

この宣言は, shades列挙型の light変数を定義しています。light は任意の列挙型値として扱うことができます。

shadesタグは新しい型の列挙型タグです。off から bright は0から4の値を持つ列挙定数です。この列挙定数は定数値であり, 整数定数が有効な場合にはいつでも使用できます。

いったんタグが宣言されると, 次を示す宣言のようにその列挙型の参照として使用できます。ここでは, light1変数は shades列挙データ型のオブジェクトになります。

```
enum shades light1;
```

列挙型に対して不完全型宣言を行うことはできません。次にその例を示します。

```
enum e;
```

enumタグは, 他の名前空間にある同じプログラム中の他の識別子と同じ名前でも構いません。ただし, enum定数名は変数および関数と同じ名前空間を共有するので, 不明瞭にならないように固有の名前を付ける必要があります。

内部的には、各列挙定数は 1 つの整数定数と関連しています。省略時の設定では、コンパイラは最初の列挙定数に 0 を割り付け、後に続く列挙定数には 1 ずつ増加した値を割り付けます。それぞれの列挙定数には任意の整数定数値を設定できます。このような構成要素の後に続く列挙定数は (特定値に設定されていない限り)、前の値よりも大きい値を受け取ります。次にその例を示します。

```
enum spectrum
{
    red, yellow = 4, green, blue, indigo, violet
} color2 = yellow;
```

この宣言では、red, yellow, green, blue, ... にそれぞれ値 0, 4, 5, 6, ... を割り付けます。列挙定数に重複する値を代入することも許可されています。

color2 の値は整数 (4) であり、“red” または “yellow” のような文字列ではありません。

4.6 ポインタの宣言

ポインタは、オブジェクトまたは関数のメモリ・アドレスを含んでいる変数です。ポインタ変数は、アスタリスク記号およびそのポインタが示すオブジェクトのデータ型を使用することにより、ポインタ型として宣言されます。構文は次のとおりです。

ポインタ:

```
* 型修飾子並びopt
* 型修飾子並びopt ポインタ
```

型修飾子並び:

```
型修飾子
型修飾子並び 型修飾子
```

宣言

4.6 ポインタの宣言

省略時の設定では、*Compaq C*のポインタのビット長は、*OpenVMS*システムで使用する場合には 32 ビットであり、*Tru64 UNIX*システムで使用する場合には 64 ビットです。省略時の設定が異なっているとしても、*OpenVMS Alpha*と*Tru64 UNIX*の両方のシステムで 32 ビット (short) と 64 ビット (long) のポインタをサポートしています。*Compaq C*では、ポインタのサイズを制御するための修飾子/スイッチや`#pragma`前処理命令が用意されています。

型修飾子は `const`、`volatile`、`__unaligned` (*Alpha*)または`__restrict`のいずれか、あるいはそれを組み合わせたものです。

ポインタ型のオブジェクトの宣言は、次の例のとおりです。

```
char *px;
```

この例では、`px`識別子を `char`型のオブジェクトへのポインタとして宣言しています。この例では、型修飾子は使用していません。式 `*px` は、`px` が示す `char` を導きます。

次に示す宣言は、定数への変数ポインタ、変数への定数ポインタ、および定数オブジェクトへの定数ポインタ間の相違を示しています。

```
const int *ptr_to_constant;    /* pointer variable pointing
                               to a const object          */
int *const constant_ptr;      /* constant pointer to a
                               non-const object           */
const int *const constant_ptr; /* Const pointer to a
                               const object                */
```

`ptr_to_constant` が示すオブジェクトの定数はこのポインタで変更することはできませんが、`ptr_to_constant`自体は別の `const`修飾オブジェクトを示すように変更することができます。これに対して、`constant_ptr` が示す整数の定数は変更できませんが、`constant_ptr`自体は常に同じ位置を示します。

`constant_ptr`定数ポインタの宣言は、`int` へのポインタ型の定義を含めることにより明確にできます。次の例では、`constant_ptr` を `int` への `const`修飾ポインタ型を持つオブジェクトとして宣言しています。ポインタの値 (アドレス) は定数です。

```
typedef int *int_ptr;  
const int_ptr constant_ptr;
```

`__unaligned`データ型修飾子は、Alpha システム上でポインタ定義に使用できます。この修飾子は、ポイントしているデータが正しいアドレスで適切に境界調整されていないことをコンパイラに指示します。適切に境界調整するには、オブジェクトのアドレスはその型のサイズの倍数でなければなりません。たとえば、2 バイトのオブジェクトは偶数のアドレスで境界調整する必要があります (*Alpha*)。

`__unaligned`と宣言したポインタでデータをアクセスすると、コンパイラはデータをコピーまたは格納するために必要な追加のコードを生成して、境界調整のエラーが出ないようにします。境界調整の誤ったデータは一切使用しないのが良いのですが、パックされた構造体にアクセスする必要やその他の理由から、使用される場合があります (*Alpha*)。

`__restrict`型修飾子は、ポインタが明確なオブジェクトを指していることを示すために使用され、コンパイラによる最適化が行われるようにします (第 3.7.4 項を参照)。

`extern` または `static`ポインタ変数は、明示的に初期化されない限り空ポインタに初期化されます。空ポインタとは、値が 0 のポインタです。初期化されていない場合の `auto`ポインタの定数は定義されていません。

4.6.1 void ポインタの宣言

`void`ポインタは指定されたデータ型を持たないポインタであり、そのポインタが示すオブジェクトを記述します。実際にはこれが汎用ポインタと呼ばれます。ANSI C 規格が制定される前は、汎用ポインタを定義するために `char*` が使用されていました。ただし、この方法は移植性が低いため、ANSI 規格では現在推奨していません。

任意の型へのポインタはキャストせずに `void`ポインタに代入することが可能であり、また、その逆も可能です。キャスト演算子については、第 6.4.6 項を参照してください。次の文では、明示的にキャストすることなく `void`ポインタを他の型のポインタに代入する方法を示しています。

宣言

4.6 ポインタの宣言

```
float *float_pointer;
void *void_pointer;
.
.
.
float_pointer = void_pointer;
/* or, */
void_pointer = float_pointer;
```

仮引数値または返却値が型の不明なポインタである場合，voidポインタは関数呼出し，関数の実引数，または関数プロトタイプで使用されます。次の例では，voidポインタが汎用返却値として使用されています。

```
void *memcpy (void *s1, const void *s2, size_t n);
{
    void *generic_pointer;
    .
    .
    .
    /* The function return value can be a pointer to many types. */
    generic_pointer = func_returning_pointer( arg1, arg2, arg3 );
    .
    .
    .
    /* size_t is a defined type */
}
```

関数宣言に voidを使用する方法については，第 5.3 節を参照してください。

4.6.2 ポインタの初期化

ポインタ・オブジェクトは単式で初期化できます。次にその例を示します。

```
int i = 10;
int *p = &i; /* p is a pointer to int, initialized */
             /* as holding the address of i */
```

初期化子がない場合には，static および externポインタの値は自動的に空ポインタ（メモリ位置 0 へのポインタ）に初期化されます。

次に示す宣言は `char` へのポインタ型で `p` を定義し、`p` を初期化して、長さ 4 の `char` 型配列のオブジェクトを示すようにします。このオブジェクトの各要素は、文字列リテラルによって初期化されます (ヌル文字は配列の 4 番目のメンバです)。配列の定数を修正するために `p` を使用した場合の動作結果は定義されていません。

```
char *p = "abc";
```

4.7 配列の宣言

次に示す構文のように、配列は大括弧 `[]` で宣言します。

記憶域クラス指定子_{opt} 型指定子 宣言子 [*または 定数式並び_{opt}]

次の例は、10 個の整数の要素を持つ配列の宣言を示します。 `table_one` という変数を使用しています。

```
int table_one[10];
```

型指定子は要素のデータ型を示しています。配列の各要素は、任意のスカラー・データ型または集合体データ型の要素になります。 `table_one` 識別子は配列名を指定します。定数式の 10 は、1 次元での要素数になります。C 言語での配列は 0 が基底です。つまり、次の例に示すように配列の最初の要素は 0 の添字で識別されます。

```
int x[5];
x[0] = 25; /* The first array element is assigned the value 25 */
```

宣言中の大括弧で囲んだ式は、区切り子 `(*)`、または 0 よりも大きい値を持つ整数定数式でなければなりません。

`*` を大括弧で囲んで指定すると、配列の型はサイズを指定しない可変長配列型になります。この型は、関数プロトタイプ・スコープの宣言でのみ使用できます。

サイズの式が整数定数式で、要素の型が既知の定数サイズの場合、配列型は可変長配列型にはなりません。それ以外の場合は、可変長配列型になります。可変長配列型の各インスタンスのサイズは、その存在期間の間変更されません。可変長配列についての詳細は、第 4.7.3 項を参照してください。

宣言

4.7 配列の宣言

*または定数式を省略すると，不完全な配列宣言が作成されます。これは，次の場合に役に立ちます。

- 配列が外部宣言され，その記憶域が他の場所での定義によって割り付けられる場合には，配列名を次の例のように宣言すると，便宜上，定数式を省略することができます。

```
extern int array1[];
int first_function(void)
{
    .
    .
    .
}
```

別のコンパイル単位で次の宣言を行います。

```
int array1[10];
int second_function(void)
{
    .
    .
    .
}
```

多次元配列宣言の最初の大括弧からのみ，配列サイズ指定子を省略できます。これは，配列自体が不完全型を持っていたとしても，配列の要素は完全型を持たなければならないためです。

- 配列の宣言に初期化子 (第 4.7.1 項および第 4.9 節を参照) を含める場合には，配列のサイズを省略できます。次にその例を示します。

```
char array_one[] = "Shemps";
char array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };
```

この 2 つの定義は同じ要素で変数を初期化します。この配列は，7 つの要素 (6 つの文字と文字列を終了させるヌル文字 '\0') を持っています。配列のサイズは，初期化文字列定数または初期化並びの文字数によって決定されます。不完全な配列を初期化すると配列型が完了します。配列は初期化子並びの終わりで完了します。

- 関数の仮引数として配列を使用する場合は、配列を呼出し関数に定義しなければなりません。ただし、呼び出される関数の仮引数宣言では、大括弧内の定数式を省略できます。配列の最初の要素のアドレスが渡されます。呼出し関数で添字を付けて参照すると、配列の要素を修正できます。次の例では、このような配列の使用方法を示しています。

```
main()
{
    /* Initialize array */
    static char arg_str[] = "Thomas";
    int sum;
    sum = adder(arg_str); /* Pass address of first array element */
    .
    .
    .
}

/* adder adds ASCII values of letters in array */
int adder( char param_string[])
{
    int i, sum = 0; /* Incrementer and sum */
    /* Loop until NULL char */
    for (i = 0; param_string[i] != '\0'; i++)
        sum += param_string[i];
    return sum;
}
```

adder関数が呼び出されると、param_string仮引数は arg_str実引数の最初の文字のアドレスを受け取り、それを adder でアクセスすることができます。param_string の宣言は仮引数の型を与えるだけで、その記憶域は確保しません。

また、配列のメンバはポインタでも構いません。次の例は、浮動小数点数の配列と浮動小数点数のポインタの配列を宣言する例です。

```
float fa[11], *afp[17];
```

宣言

4.7 配列の宣言

関数の仮引数を配列として宣言すると、コンパイラはその宣言を配列の最初の要素へのポインタとみなします。たとえば、`x` が仮引数であり、整数の配列を表すことを目的としている場合には、次に示す宣言のいずれかで宣言することができます。

```
int x[];  
int *x;  
int x[10];
```

関数の仮引数の場合には、配列に指定するサイズはどのような長さでも構わないことに注意してください。これは、ポインタが常に配列の最初の要素のみを示すためです。

C 言語は、配列の配列として宣言された配列をサポートしています。これを多次元配列と呼びます。次の例では、`table_one` 変数は 20 の整数を含む 2 次元配列になっています。

```
int table_one[10][2];
```

配列は行優先順序で格納されます。この例の場合、`table_one[0][0]` 要素の直後に `table_one[0][1]` が続き、その直後に `table_one[1][0]` が続きます。

4.7.1 配列の初期化

配列は、中括弧で囲んだ定数式並びによって初期化されます。不完全配列宣言の初期化子並びは配列型を完了し、配列のサイズを完全に定義します。したがって、不明のサイズの配列を初期化すると、初期化子並びの初期値の数が配列のサイズを決定します。たとえば、次に示す宣言は、3 つの要素の配列を初期化します。

```
int x[] = { 1, 2, 3 };
```

初期化する配列が `static` 記憶域クラスである場合には、初期化子は定数式でなければなりません。

サイズを指定した配列の初期化子は、1 対 1 でそれぞれの配列メンバに代入されます。メンバに対して初期化子が少ない場合には、残りのメンバは 0 に初期化されます。指定したサイズ配列に対して多すぎる初期化子を表示すると、エラーになります。次にその例を示します。


```
int x[5] = { 0, 1, 2, 3, 4, 5 };    /* error    */
```

文字列リテラルは、char または wchar_t 配列に代入される場合があります。この場合、文字列の各文字は 1 次元配列の 1 つのメンバを表しており、その配列はヌル文字で終了します。配列を文字列リテラルへのポインタによって初期化した場合は、その文字列リテラルをそのポインタで変更することはできません。

文字列リテラルで配列を初期化する場合、初期化する文字列を二重引用符で囲みます。次にその例を示します。

```
char string[26] = { "This is a string literal." };  
/* The braces above are optional here */
```

終了ヌル文字はサイズに余裕があれば、この例に示したように文字列の終わりに追加されます。文字を含む配列を初期化するための別の形式は、次のとおりです。

```
char string[12] = { 'T', 'h', 'i', 's', ' ', 'w', 'a', 'y' };
```

この例では、文字列の値 “This way” を含む 1 次元の配列を作成しています。この配列の文字は自由に変更できます。初期化されない残りの配列メンバは、自動的に 0 に初期化されます。

文字列リテラルに使用する配列のサイズを明示的に記述していない場合には、そのサイズは文字列の文字数 (終了ヌル文字を含む) によって決まります。配列のサイズを明示的に記述している場合には、その配列よりも長い文字列リテラルで配列を初期化するとエラーになります。

注意

ヌル文字は、自動的に配列に追加されない場合があります。つまり、配列サイズを明示的に指定した場合に、初期化子の数で配列サイズが満杯になった場合です。次にその例を示します。

```
char c[4] = "abcd";
```

この場合、c 配列は 4 つの指定文字である a, b, c, d のみを保持します。したがって、ヌル文字でこの配列が終了することはありません。

宣言

4.7 配列の宣言

多次元配列のメンバを初期化する場合には、次に示す規則に従って中括弧を省略できます。

- 配列の初期化時には一番外側の中括弧を省略できる。
- 初期化子並びに初期化するオブジェクトの初期化子がすべて含まれている場合には、内側の中括弧を省略できる。

次にその例を示します。

```
float x[4][2] = {  
    { 1, 2 }  
    { 3, 4 }  
    { 5, 6 }  
};
```

この例では、1 と 2 が x 配列の最初の行を初期化し、それに続く 2 つの行が 2 番目と 3 番目の行をそれぞれ初期化します。この初期化は、4 番目の行が初期化される前に終わるので、4 番目の行のメンバは省略時の設定である 0 になります。結果は次のとおりです。

```
x[0][0] = 1;  
x[0][1] = 2;  
x[1][0] = 3;  
x[1][1] = 4;  
x[2][0] = 5;  
x[2][1] = 6;  
x[3][0] = 0;  
x[3][1] = 0;
```

次に示す宣言でも、前の例と同じ結果になります。

```
float x[4][2] = { 1, 2, 3, 4, 5, 6 };
```

この場合、コンパイラは配列を 1 行ずつ使用可能な初期値で満たします。コンパイラは、1 と 2 を最初の行 (x[0]) に、3 と 4 を 2 番目の行 (x[1]) に、5 と 6 を 3 番目の行 (x[2]) に入れます。配列の残りのメンバは 0 で初期化されます。

注意

- 配列および構造体にディジグネーションを使用した初期化子の説明については、第 4.9 節を参照してください。
 - 可変長配列は初期化できません。
-

4.7.2 ポインタと配列

配列のデータ・オブジェクトは、配列添字を使用する代わりにポインタで参照できます。このポインタのデータ型は、「配列へのポインタ型」として参照されます。配列名自体はポインタと同様に機能するため、配列要素へのアクセスにはいくつかの代替方法があります。次にその例を示します。

```
int x[5] = { 0, 1, 2, 3, 4 }; /* Array x declared with five elements */
int *p = x;                  /* Pointer declared and initialized to point */
                              /* to the first element of the array x */

int a, b;
a = *(x + 3);                /* Pointer x incremented by twelve bytes */
                              /* to reference element 3 of x */
b = x[3];                    /* b now holds the same value as a */
```

この例では、`a` は逆参照演算子 (`*`) を使用して 3 を受け取ります。`b` は添字演算子を使用して、同じ値を受け取ります。それぞれの単項演算子についての詳細は、第 6 章を参照してください。

`a` の代入は、`x` へのポインタが増分された結果であることに注意してください。これをスケーリングといい、ポインタ演算のすべての型に適用されます。スケーリングでは、配列メンバのメモリ・アドレスの計算時にコンパイラが配列要素のサイズを考慮します。たとえば、`x` 配列の各メンバが 4 バイト長の場合に初期ポインタ値に 3 を加算すると、この 3 に配列メンバのサイズ (この場合は 4) を乗算した値に自動的に変換されます。これによって、`z = *(y + 3);` というポインタ算術を理解しやすく表現することができます。

関数の実引数として配列を渡す場合には、配列の最初の要素へのポインタのみが呼び出される関数に渡されます。配列型からポインタ型への変換は、暗黙に行われます。配列名が配列の最初の要素へのポインタに変換された後は、他のポインタと同

宣言

4.7 配列の宣言

様にポインタを増分，減分，または逆参照してその配列内のデータを処理できます。次にその例を示します。

```
int func(int *x, int *y) /* The arrays are converted to pointers      */
{
    *y = *(x + 4);        /* Various elements of the arrays are accessed */
}
```

ポインタはアドレスを保持するだけの大きさです。すなわち，配列へのポインタは，その配列の要素のアドレスのみを保持します。配列自体は配列のメンバすべてを保持するのに十分な大きさです。

sizeof演算子を配列に適用すると，配列の最初の要素のサイズではなく，配列全体のサイズを返します。

4.7.3 可変長配列

可変長配列を使用すると，auto記憶域クラスの配列オブジェクトと，ブロック・スコープで宣言されているtypedef配列は，実行時に計算される式を境界とすることができます。

また，可変長配列を使用すると，他の仮引数で次元が指定される配列 (FORTRAN の assumed-shape 配列のようなもの) を仮引数に持つ関数の宣言と定義ができます。

次の例では，その両方の使用方法を示します。sub関数の定義でプロトタイプ構文が使用されていることと，次元仮引数が，それを使用する配列仮引数より前にあることに注意してください。次元仮引数の後に，それを使用する配列仮引数が続く関数を定義する場合は，Kernighan と Ritchie の C の構文を使用して関数定義を記述しなければなりません (この構文では，仮引数の型の宣言を仮引数自体とは異なる順序で記述できるためです)。このような関数定義は，一般的には避けてください。

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

void sub(int, int, int[*][*]);

int main(int argc, char **argv)
{
    if (argc != 3) {
        printf("Specify two array bound arguments.\n");
        exit(EXIT_FAILURE);
    }
    {
        int dim1 = atoi(argv[1]);
        int dim2 = atoi(argv[2]);
        int a[dim1][dim2];
        int i, j, k = 0;
        for (i = 0; i < dim1; i++) {
            for (j = 0; j < dim2; j++) {
                a[i][j] = k++;
            }
        }
        printf("dim1 = %d, dim2 = %d.",
               sizeof(a)/sizeof(a[0]),
               sizeof(a[0])/sizeof(int));

        sub(dim1, dim2, a);
        sub(dim2, dim1, a);
    }
    exit(EXIT_SUCCESS);
}

void sub(int sub1, int sub2, int suba[sub1][sub2])
{
    int i, j, k = 0;
    printf("\nIn sub, sub1 = %d, sub2 = %d.",
           sub1, sub2);
    for (i = 0; i < sub1; i++) {
        printf("\n");
        for (j = 0; j < sub2; j++) {
            printf("%4d", suba[i][j]);
        }
    }
}
```

OpenVMS システムでは、非標準の `alloca intrinsic` , `__ALLOCA` の代わりに可変長配列を使用することがよくあります。

ただし、`__ALLOCA` と可変長配列には重要な違いがあります。`__ALLOCA` によって割り当てられた記憶域は関数から戻るまで解放されませんが、可変長配列によって割り当てられた記憶域は、それを割り当てたブロックを終了した時点で解放されるという点です。`__ALLOCA` が、可変長配列宣言のスコープ内部で呼び出された場合 (可変長配列の宣言を含むブロックの内部でネストされているブロックも含む)、`__ALLOCA` への呼出しによって割り当てられた記憶域は、その可変長配列の記憶域が解放された時点で解放されます (つまり、関数から戻る時点ではなく、ブロックの終了時に解放されます)。このような場合、コンパイラは警告を出力します。

4.8 構造体と共用体の宣言

構造体は一連のメンバで構成されており、その各メンバには記憶域が順に割り付けられます。これに対して、共用体も一連のメンバで構成されていますが、その各メンバの記憶域は重複しています。構造体と共用体の宣言は、次に示すように同じ形式です。

構造体または共用体指定子:

```
struct または union 識別子opt { 構造体宣言並び }
struct または union 識別子
```

struct または union:

```
struct
union
```

構造体宣言並び:

```
構造体宣言
構造体宣言並び 構造体宣言
```

構造体宣言:

```
指定子修飾子並び 構造体宣言子並び ;
```

指定子修飾子並び:

型指定子 指定子修飾子並び_{opt}
型修飾子 指定子修飾子並び_{opt}

構造体宣言子並び:

構造体宣言子
構造体宣言子並び , 構造体宣言子

構造体宣言子:

宣言子
宣言子_{opt} : 定数式

構造体および共用体は、関数型または不完全型を持つことはできません。構造体および共用体はそれ自体のインスタンスをメンバとして追加することはできませんが、それ自体のインスタンスへのポインタを追加することはできます。メンバを持たない構造体の宣言であれば受け入れ可能であり、そのサイズは0です。

各構造体または共用体は、コンパイル単位内に固有の構造体型または共用体型を作成します。struct または union キーワードをタグの前に付けることができます。enum タグが列挙型に名前を与えるのと同様に、このタグは構造体型または共用体型に名前を与えます。このタグを struct または union キーワードと一緒に使用すれば、長い定義を繰り返さずにその型の変数を宣言できます。

タグの後には、メンバ宣言並びを囲む中括弧 { } を続けます。並びの各宣言にはデータ型と1つ以上のメンバ名を指定します。構造体または共用体メンバの名前は、他の変数、関数名、他の構造体メンバ、または共用体のメンバと同じにすることができます。コンパイラは、使用状況に応じてこれらを区別します。メンバ名のスコープは、使用している構造体または共用体のスコープと同じです。中括弧を閉じて並びを完了すると、構造体型または共用体型は完了します。

構造体タグまたは共用体タグに使用する識別子は、そのスコープ内で参照できるタグの中で固有でなければなりません。ただし、タグ識別子は、変数名または関数名に使用する識別子と同じにすることができます。また、タグはメンバ名と同じ名前にすることもできます。コンパイラは、それらを名前空間と使用状況で区別します。タグのスコープは、使用している宣言のスコープと同じです。

宣言

4.8 構造体と共用体の宣言

構造体と共用体には、他の構造体と共用体を含めることができます。次にその例を示します。

```
struct person
{
    char first[20];
    char middle[3];
    char last[30];
    struct      /* Nested structure here */
    {
        int day;
        int month;
        int year;
    } birth_date;
} employees, managers;
```

構造体宣言または共用体宣言は、次に示す形式のいずれかにすることができます。

- 宣言にタグとメンバ宣言並びのみを含めると、メンバ宣言並びはそのタグを他のオブジェクトを宣言できるデータ型として定義します。タグは、その構造体型の短縮表記とみなされます。次にその例を示します。

```
struct person
{
    char first[20];
    char middle[3];
    char last[30];
};
struct person employee; /* The tag (person) identifies employee as */
                        /* a structure with members shown in */
                        /* the declaration of person */
```

- 宣言にタグ、メンバ宣言並び、および識別子並びを含めると、その識別子が構造体型のオブジェクトになり、そのタグは構造体型の短縮表記つまり簡略表記とみなされます。次にその例を示します。

```
struct person
{
    char first[20];
    char middle[3];
    char last[30];
} employees, managers;
```


- タグを省略すると、構造体定義または共用体定義は宣言中の識別子にのみ適用されます。次にその例を示します。

```
struct
{
    char first[20];
    char middle[3];
    char last[30];
} employees, managers;
```

- タグは、別の場所に定義されている構造体または共用体を参照できます。さらに、その定義は宣言中のタグ名の後続く変数識別子に適用されます。次にその例を示します。

```
struct person employees, managers;
```

- struct または union キーワードとタグのみを使用して、スコープ内にある同じ名前の他のタグを上書きし、後で新しいスコープ内で定義するためにそのタグを確保することができます。新しいスコープ内のタグ定義は、外側のスコープに指定されている同じタグ定義を上書きします。このタグ宣言の使用を構造体タグの仮宣言と呼びます。この宣言を使用することにより、タグ識別子の順方向参照を行う際の不明瞭さがなくなります。次にその例を示します。

```
struct A {...}; /* Definition of external struct A */
{
    struct A; /* Tentative structure tag declaration. */
              /* First declaration of A (in external scope) is
              hidden. This structure will be defined later */
    struct inner
    {
        struct A *pointer; /* Declare a structure pointer by */
                          /* forward referencing. */
        .
        .
        .
    };
    struct A {...}; /* Tentative declaration of internal struct A is
                     defined here. */
                  /* External struct A is unaffected by this definition*/
}
```

この例では、Aタグを使用して定義された構造体のポインタは外部定義ではなくAの内部定義を示します。

4.8.1 構造体と共用体の類似点

構造体と共用体には、次のような共通の特性があります。

- メンバは、他の構造体や共用体または配列を含むどのような型のオブジェクトにすることもできる。また、メンバはビット・フィールドで構成することもできる。
- 構造体と共用体全体で使用する有効な演算子は、単純代入演算子(=)およびsizeof演算子だけである。特に、構造体と共用体は等価演算子(==)、非等価演算子(!=)、またはキャスト演算子のオペランドとして指定することはできない。代入における2つの構造体または共用体は、同じメンバとメンバ型を持たなければならない。
- 構造体または共用体は関数との間で値を渡したり、渡されたりする。実引数は、関数の仮引数と同じ型を持たなければならない。構造体または共用体は、スカラー変数と同様に値が渡される。つまり、構造体または共用体全体がそれに対応する仮引数にコピーされる。

注意

構造体を実引数として渡すとき、ロングワードの境界上で終了する場合と終了しない場合とがあります。終了しない場合は、次のロングワード境界上で後続の実引数の境界調整を行います。

4.8.2 構造体と共用体の相違点

構造体と共用体の相違点は、メンバの格納および初期化の方法にあります。次に相違点について説明します。

- 構造体の各メンバのアドレスは、左から右へ宣言を読み込むにつれて増加する。つまり、構造体のすべてのメンバは構造体の基底とは異なるオフセットか

ら始まる。特定メンバのオフセットは宣言が読み込まれた順序に応じて異なり、最初のメンバはオフセット 0 である。

構造体へのポインタはその最初のメンバを示す。したがって、名前がないホールが構造体の始めに存在することはない。

*OpenVMS VAX*システム上では、ビット・フィールド以外の構造体メンバは、省略時の設定ではバイト境界で調整される。ただし、`#pragma [no]member_alignment`と`#pragma pack`の前処理命令が用意されており、バイト境界調整から自然境界調整に切り替えることができる。

Alpha システム上では、ビット・フィールド以外の構造体メンバは自然に境界調整され、ビット・フィールド以外の後続の構造体メンバはその型に適した境界調整に一致する次のバイト境界から始まる。たとえば、短整数 (`short`) は 2 バイト境界で調整され、ロング整数 (`long`) は 4 バイト境界で調整される。したがって、構造体には名前がないホールが存在する可能性がある。

Alpha プロセッサ上で自然に境界調整された構造体の長さは、任意のメンバの最大の境界調整条件の倍数でなければならない。たとえば、文字、短整数、ロングワードを含む構造体は、ロングワード用の 4 バイトの倍数と一致するように長さが 4 の倍数になる。

`#pragma [no]member_alignment`と`#pragma pack`の前処理命令は、このプラットフォーム上でもサポートされる。

特殊な構造体の境界調整条件および例については、プラットフォームに固有の *Compaq C* マニュアルを参照すること。

- 共用体の各メンバは、共用体の開始アドレスからオフセット 0 のところで始まる。メモリ内の共用体のサイズは、サイズの最も大きいメンバと同じである。共用体のオブジェクトには、一度に 1 つのメンバの値しか格納できない。共用体に割り付けられる記憶域に小さい方のメンバを入れると、その小さい方のメンバの終わりから割り付けられたメモリの終わりまでの間に余分な領域ができ、そのまま変更されずに残る。共用体メンバの境界調整に関する規則は、構造体メンバの規則と同様である (プラットフォームに固有の *Compaq C* のマニュアルを参照のこと)。

共用体メンバへのポインタは適切な型に変換され、共用体オブジェクトの始まりを示す。

- 構造体では複数のメンバを一度に初期化できる。これに対して、共用体では最初のメンバのみに初期値を与えることができる。

4.8.3 ビット・フィールド

構造体の利点の 1 つは、構造体にビット単位でデータをパックできることです。

構造体メンバは通常、基本型サイズを持つオブジェクトです。ただし、指定したビット数のみで構成される構造体メンバを宣言することもできます。このようなメンバをビット・フィールドと呼びます。次の構文に示すように、ビット・フィールドの長さ、つまり負以外の汎整数定数式はコロンでフィールド名と区切られます。

構造体宣言子:

```
宣言子: 定数式  
      : 定数式
```

ビット・フィールドによって、構造体の記憶域割付けの制御が向上し、メモリ内に情報をより緊密にパックすることができます。また、データを記憶域に高密度でパックすることができます。

ビット・フィールド型は、名前のないビット・フィールドの場合以外は指定する必要があります。また、ビット・フィールド型は `int`、`unsigned int`、または `signed int` 型を持つことができます。ビット・フィールド値は、宣言したサイズのオブジェクトに格納できるだけの小さい値にする必要があります。

コンパイラの省略時のモードでは、`enum`、`long`、`short`、および `char` 型もビット・フィールドに使用できます。

ビット・フィールドには名前を付けても付けなくても構いません。宣言子を持たないビット・フィールド宣言 (例: `:10`) は、名前のないビット・フィールドを示します。これは、指定のレイアウトに合わせるために、構造体をパディングするときに有効です。ビット・フィールドに幅 0 を割り付けると、その境界調整単位にはそれ以上ビット・フィールドが置かれないことを示し、宣言子を命名することはできません。メンバの宣言子 (存在する場合) とフィールド幅をビット単位で示す定数式とを区切るには、コロン `:` を使用します。フィールドは、32 ビット (1 ロングワード) 以下でなければなりません。

非ビット・フィールドの構造体メンバはバイト境界で調整されるため、名前のない形式は構造体の記憶域に名前のない空きをつくることができます。特殊な場合として、幅 0 の名前のないフィールドの次のメンバ (通常は別のフィールド) は、バイト境界で境界調整されます。つまり、幅が 0 のビット・フィールドの構造体メンバは、それ以上ビット・フィールドを境界調整単位にパックしてはならないということになります。

ビット・フィールドを使用する際に適用される規則は、次のとおりです。

- ビット・フィールドの配列は宣言できない。
- アンパサンド演算子 (&) をフィールドに適用できないため、ビット・フィールドへのポインタを設定することができない。

ビット・フィールドの列はできるだけ緊密にパックされます。 *Compaq C* ではビット・フィールドは右から左に、つまり下位ビットから上位ビットに割り付けられます。

ビット・フィールドを作成するには、構造体メンバとして識別子、コロン、および識別子幅 (ビット単位) を指定します。次の例では、3 つのビット・フィールドが構造体宣言に作成されます。

```
struct {  
    unsigned int a : 1; /* Named bit field (a) */  
    unsigned int : 0; /* Unnamed bit field = 0 */  
    unsigned int : 1; /* Unnamed bit field */  
} class;
```

最初と 3 番目のビット・フィールドは 1 ビット幅であり、2 番目のビットは 0 ビット幅です。これによって、次のメンバはバイト境界上に境界調整されます。

他のビット・フィールドの直後に宣言されなかったビット・フィールド (長さ 0 のビット・フィールドを含む) は、それらの型によって強制される境界調整条件を持ちます。この条件は、int の境界調整条件よりは必ず大きくなります。他のビット・フィールドの直後に続くビット・フィールドの宣言では、領域が十分に残っていれば同じ境界調整単位の隣接する領域にそのビットがパックされます。これ以外の場合はパディングが挿入され、2 番目のビット・フィールドは次の境界調整単位に置かれます。

構造体内のビット・フィールド境界調整の情報に関するプラットフォーム固有の情報については、該当する *Compaq C* のマニュアルを参照してください。

4.8.4 構造体の初期化

すべての構造体は、構成要素の初期値を中括弧で囲んだ並びで初期化できます。また、自動記憶域クラスを持つ構造体は、互換型の式で初期化することもできます。

初期化子は 1 対 1 で構成要素に代入されます。構造体のメンバよりも初期化子が少ない場合には、残りのメンバは 0 に初期化されます。構成要素の数に対して構造体に表示する初期化子が多すぎると、エラーが生じます。名前のない構造体または共用体のメンバはすべて、初期化時には無視されます。

コンマで初期値を区切り、中括弧 { } で範囲を決定します。次の例では、それぞれ 2 つのメンバを持つ 2 つの構造体を初期化しています。

```
struct
{
    int i;
    float c;
} a = { 1, 3.0e10 }, b = { 2, 1.5e5 };
```

コンパイラは、各メンバの増加順に構造体の初期化子を代入します。構造体の中央にあるメンバは、それ以前のメンバを初期化せずに初期化することはできません。

例 4-1 は、構造体の配列に適用される初期化規則の例を示しています。

例 4-1 構造体の初期化の規則

```
#include <stdio.h>

main()
{
    int m, n;
    static struct
    {
        char ch;
        int i;
        float c;
    } ar[2][3] =
1      {
2      {
3      { 'a', 1, 3e10 },
        { 'b', 2, 4e10 },
        { 'c', 3, 5e10 },
      }
    };

    printf("row/col\t ch\t i\t\t c\n");
    printf("-----\n");
    for (n = 0; n < 2; n++)
        for (m = 0; m < 3; m++)
        {
            printf("[%d][%d]:", n, m);
            printf("\t %c\t %d\t %e\n",
                ar[n][m].ch, ar[n][m].i, ar[n][m].c);
        }
}
```

例 4-1 の説明。

- 1 中括弧で配列行の初期化の範囲を指定します。
- 2 中括弧で構造体の初期化の範囲を指定します。
- 3 中括弧で配列の初期化の範囲を指定します。

宣言

4.8 構造体と共用体の宣言

例 4-1 では、次の出力を標準出力に書き込みます。

row/col	ch	i	c
[0] [0]:	a	1	3.000000e+10
[0] [1]:	b	2	4.000000e+10
[0] [2]:	c	3	5.000000e+10
[1] [0]:		0	0.000000e+00
[1] [1]:		0	0.000000e+00
[1] [2]:		0	0.000000e+00

注意

配列および構造体にディジグネーションを使用した初期化子の説明については、第 4.9 節を参照してください。

4.8.5 共用体の初期化

共用体は、中括弧で囲んだ初期化子で初期化されます。この初期化子は、共用体の最初のメンバのみを初期化します。次にその例を示します。

```
static union
{
    char ch;
    int i;
    float c;
} letter = {'A'};
```

auto記憶域クラスを持つ共用体は、その共用体と同じ型の式でも初期化が可能です。次にその例を示します。

```
main ()
{
    union1 {
        int i;
        char ch;
        float c;
    } number1 = { 2 };
```



```
auto union2
{
    int i;
    char ch;
    float c;
} number2 = number1;
}
```

4.9 デジグネーションを使用した初期化子

C99 規格に適合するように、*Compaq C* は配列および構造体の初期化でのデジグネーションの使用をサポートしています。(デジグネーションは、コンパイラのコモン C、VAX C、Strict ANSI89 モードではサポートされません。)

4.9.1 カレント・オブジェクト

C99 初期化子では、カレント・オブジェクトおよびデジグネーションという概念を導入しました。

カレント・オブジェクトは、配列あるいは構造体の初期化時に、次に初期化されるものです。

デジグネーションは、カレント・オブジェクトを設定する方法を提供します。デジグネーションが存在しない場合、カレント・オブジェクトのサブオブジェクトはオブジェクトの型に従って初期化されます。配列要素は添字の昇順に、構造体メンバは宣言順に初期化されます。

したがって、配列については、初期化が始まった際の最初のカレント・オブジェクトは `a[0]` です。各初期化子で使用されるに従い、カレント・オブジェクトは添字の昇順に次の初期化子に移行します。

同様に、構造体については、初期化が始まった際のカレント・オブジェクトは構造体内の最初の宣言です。各初期化子で使用されるに従い、カレント・オブジェクトは宣言順に次の初期化子に移行します。

宣言

4.9 デイジグネーションを使用した初期化子

4.9.2 デイジグネーション

C99 規格では、デイジグネーションを含む中括弧で囲まれた初期化子並びを使用できます。これは新しいカレント・オブジェクトを指定します。デイジグネーションの構文は次のとおりです。

デイジグネーション:

指名子並び=

指名子並び:

指名子
指名子並び 指名子

指名子:

[定数式]
.識別子

デイジグネーション内の指名子は、次の初期化子に指名子で記述されたオブジェクトの初期化を開始させます。続いて、初期化は順次継続され、指名子で記述された次のオブジェクトから始まります。

配列については、指名子は次のとおりです。

[凡整数定数式]

配列のサイズが不明である場合は、負数でない値が有効です。

構造体については、指名子は次のとおりです。

.識別子

ここで、識別子は構造体のメンバです。

4.9.3 例

配列および構造体を初期化する従来の方法は、継続してサポートされています。ただし、指名子の使用により、初期化子並びのコーディングを簡素化することができます、アプリケーション内の配列および構造体に対して行う将来的な変更に対応することができます。

1. 指名子を使用することにより、配列要素はその順序に関係なく非ゼロ値に初期化することができます。

```
int a[5] = { 0, 0, 0, 5 }; // Old way
int a[5] = { [3]=5 };      // New way
```

指名子[3]はa[3]を5に初期化します。

2. 構造体のメンバは、その順序に関係なく非ゼロ値に初期化することができます。次の例を参照してください。

```
typedef struct {
    char flag1;
    char flag2;
    char flag3;
    int data1;
    int data2;
    int data3;
} Sx;

Sx = { 0, 0, 0, 0, 6 }; // Old way
Sx = { .data2 = 6 };    // New way
```

指名子.data2は、構造体メンバの.data2を6に初期化します。

3. 次は、配列に指名子を使用する別の例です。

```
int a[10] = { 1, [5] = 20, 10 };
```

この例では、配列要素は次のように初期化されます。

宣言

4.9 デイジグネーションを使用した初期化子

```
a[0]=1
a[1] through a[4] = 0
a[5] = 20
a[6] = 10
a[7] through a[9] = 0
```

4. 構造体に対する将来的な変更にも，その初期化子並びを変更することなく対応することができます。

```
typedef struct {
    char flag1;
    char flag2;
    char flag3;
    int data1;
    int data2;
    int data3;
} Sx;
```

```
Sx = { 1, 0, 1, 65, 32, 18 }; // Old way
```

```
Sx = { .flag1=1, 0, 1, .data1=65, 32, 18 }; // New way
```

指名子 `.flag1` および `.data1` を使用することにより，`.flag1` の前あるいは `flag3` と `data1` の間への将来的なフラグの追加を可能とします。

指名子を順番に記述する必要はありません。たとえば，次の2つの初期化子並びは同等です。

```
Sx = { .data1=65, 32, 18, .flag1=1, 0, 1 };
```

```
Sx = { .flag1=1, 0, 1, .data1=65, 32, 18 };
```

5. 単一指名子を使用することにより，配列の両端からスペースを割り当てることができます。

```
int a[MAX] =
{
    1, 3, 5, 7, 9, [MAX - 5] = 8, 6, 4, 2, 0
};
```

この例では，`MAX` が 10 よりも大きい場合，中間のいくつかの要素はゼロ値を持ちます。10 よりも小さい場合，最初の初期化子で指定されたいくつかの値は，次に指定された5つの値によって上書きされます。

6. 指名子はネストすることができます。

```
struct { int a[3], b } w[] =
{ [0].a = {1}, [1].a[0] = 2 };
```

この初期化は、次と同等です。

```
w[0].a[0]=1;
w[1].a[0]=2;
```

7. ネストした指名子の別の例です。

```
struct {
    int a;
    struct {
        int b
        int c[10]
    }x;
}y = { .x = {1, .c = {[5] = 6, 7 }}}}
```

この初期化は、次と同等です。

```
y.x.b = 1;
y.x.c[5] = 6;
y.x.c[6] = 7;
```

4.10 タグの宣言

次に示す構文は、構造体タグ、共用体タグ、または列挙型タグとしてタグ識別子を宣言します。このタグ宣言が可視状態であれば、次に続くタグ参照は宣言された構造体型、共用体型、または列挙型に置き換えることができます。同じスコープ(可視状態の宣言)内のタグの次に続く参照は、括弧で囲んだ並びを省略しなければなりません。タグの構文は次のとおりです。

```
struct タグ { 宣言子並び }

union タグ { 宣言子並び }

enum タグ { 列挙子並び }
```

完全な構造体宣言または共用体宣言なしにタグが宣言されると、そのタグは不完全型を参照します。不完全な列挙型は使用できません。不完全型は、型が必要ではない場合にオブジェクトを指定するためにのみ有効です。たとえば、型定義およびポインタ宣言を行うときなどです。型を完了するには、(囲んだブロック内を除く) 同じスコープ内のタグの別の宣言が内容を定義します。

次に示す構成では、testタグを使用して自己参照構造体を定義しています。

```
struct test {  
    float height;  
    struct test *x, *y, *z;  
};
```

一度この宣言を行うと、次の宣言は struct test型のオブジェクトとして s を宣言し、struct test型のオブジェクトへのポインタとして sp を宣言します。

```
struct test s, *sp;
```

注意

typedefキーワードを使用して同じことを行うこともできます。構成は次のとおりです。

```
typedef struct test tnode;  
struct test {  
    float height;  
    tnode *x, *y, *z;  
};  
tnode s, *sp;
```

4.11 型定義の宣言

記憶域クラス指定子が typedef である宣言では、各宣言子は指定された型に別名を指定する typedef名を定義します。typedef宣言は新しい型を導入するのではなく、指定した型の同義語を導入するだけです。次にその例を示します。

```
typedef int integral_type;  
integral_type x;
```

この例では、`integral_type` は `int` の同義語として定義され、次の `x` の宣言は `int` 型として `x` を宣言しています。型定義は、長い型名 (例: 構造体や共用体の形式) を短縮形にする場合や、型定義によってその型の解釈が簡単にできる場合に有効です。

`typedef` 名は、通常の宣言の他の識別子と同じ名前空間を共有します。オブジェクトを内側のスコープに再度宣言する場合、または同じスコープか内側のスコープ内の構造体メンバや共用体メンバとして宣言する場合には、その型指定子を内側の宣言から省略することはできません。次にその例を示します。

```
typedef signed int t;  
typedef int plain;  
struct tag {  
    unsigned t:4;  
    const t:5;  
    plain r:5;  
};
```

この構成は不明瞭なものです。この例では、`signed int` 型で `t` という `typedef` 名を宣言し、`int` 型で `plain` という `typedef` 名を宣言し、3つのビット・フィールド・メンバで構造体を宣言しています。これら3つとは、`t` という名前のメンバ、名前のないメンバ、および `r` という名前のメンバです。最初の2つのビット・フィールド宣言は、1番目のビット・フィールドの型指定子が `unsigned` であるという点で異なります。前記の規則に従うと、`t` が構造体メンバの名前になります。2番目のビット・フィールド宣言には `const` 型修飾子が含まれ、これはまだ可視状態のままの `typedef` 名 `t` を修飾するだけです。

次の例は、`typedef` キーワードの別の使用方法を示しています。

宣言

4.11 型定義の宣言

```
typedef int miles, klicksp(void);
typedef struct { double re, im; } complex;
.
.
.
miles distance;
extern klicksp *metricp;
complex x;
complex z, *zp;
```

ここに示したコードはすべて有効です。distance の型は int であり, metricp の型は int を返す仮引数を持たない関数へのポインタです。x と z の型は, 指定された構造体です。zp は構造体へのポインタです。

typedef名で使用する任意の型修飾子は, 型定義の一部になるということに注意してください。そのtypedef名を同じ型修飾子で後で修飾することはできません。次にその例を示します。

```
typedef const int x;
const x y;          /* Illegal -- duplicate qualifier used */
```


Compaq C プログラムは、ユーザ定義関数とシステム定義関数の集合です。関数を使用すると大規模なタスクを小さく分割できるため、理解しやすく、かつ維持しやすいモジュール・プログラムを設計する際に役に立ちます。関数は、呼出し時に実行する 0 個以上の文を含みます。また、関数には 0 個以上の実引数を渡すこともでき、1 つの関数から 1 つの値を返すことができます。

この章では、*Compaq C* の関数について説明します。

- 関数呼出し (第 5.1 節)
- 関数型 (第 5.2 節)
- 関数定義 (第 5.3 節)
- 関数宣言 (第 5.4 節)
- 関数プロトタイプ (第 5.5 節)
- 仮引数と実引数 (第 5.6 節)

5.1 関数呼出し

関数呼出しは 1 次式であり、通常は括弧が続く関数識別子です。この識別子は関数を呼び出すために使用します。括弧の中には、コンマで区切られた式並びが含まれます。これは関数の実引数であり、空の場合もあります。次に、`power` 関数の呼出しの例を示します。この関数はすでに適切に定義されているものと仮定します。

関数

5.1 関数呼出し

```
main()
{
    .
    .
    .
    y = power(x,n);          /* function call */
}
```

関数呼出しについては、第 6.3.2 項を参照してください。

5.2 関数型

関数には、派生型である「型を返す関数」があります。この型を配列型または関数型以外のデータ型にすることはできません。配列や関数へのポインタを返すこともできます。関数が値を返さない場合、この型は「void を返す関数」になり、void 関数と呼ばれます。Compaq C の void 関数は Pascal 言語における手続き、または FORTRAN におけるサブルーチンと同じものです。Compaq C の void 関数以外の関数は、他の言語で使用する関数と同じです。

関数は、次のいずれかの方法でプログラムに使用することができます。

- 関数定義は関数指名子の作成、仮引数と型の定義、および返却値の型の定義を行い、さらに関数の本体を提供します。次の例で、power は int を返す関数です。

```
int power(int base, int exp)
{
    int n=1;
    if (exp < 0)
    {
        printf ("Error: Cannot handle negative exponent\n");
        return -1;
    }
    for ( ; exp; exp--)
        n = base * n;
    return n;
}
```

関数定義については，第 5.3 節を参照してください。

- 関数宣言は，別の場所に定義されている関数の特性を示します。次の例では main 関数が power 関数を宣言し，呼び出しています。関数定義，つまりコードが定義されている場所は別に存在します。

```
main()
{
    int power(int base, int exp);    /* function declaration */
    int x, n, y;
    .
    .
    .
    y = power(x,n);                 /* function call          */
}
```

仮引数が仮引数型並びに宣言される関数定義の形式は，関数プロトタイプと呼ばれます。関数プロトタイプでは，コンパイラは関数の実引数が各仮引数と一致するかどうかを検査し，その実引数を仮引数の宣言された型に変換します。

関数宣言とプロトタイプについては，第 5.4 節と第 5.5 節を参照してください。

5.3 関数定義

関数定義には関数のコードが含まれます。関数定義はどのような順序でも記述できます。また，1 つの関数を複数のファイルに分割することはできませんが，1 つまたは複数のソース・ファイルのいずれにでも記述できます。関数定義はネストすることはできません。

関数定義の構文は次のとおりです。

関数定義:

宣言指定子_{opt} 宣言子 宣言並び_{opt} 複文

関数

5.3 関数定義

宣言指定子

宣言指定子(記憶域クラス指定子, 型修飾子, および型指定子) は, 順不同で指定することができます。すべてオプションです。

記憶域クラス指定子は特に指定しない限り, `extern` となります。 `static` 指定子を使用することもできます。記憶域クラス指定子については, 第 2.10 節を参照してください。

ANSI 規格では, 型修飾子を `const` または `volatile` に指定することもできますが, 関数の返す型にこの修飾子を使用しても意味がありません。これは, 関数は右辺値のみを返し, 型修飾子は左辺値のみに適用されるからです。

型指定子は関数が返す値のデータ型です。返却値の型が指定されていない場合, その関数は `int` 型の値を返すものとして宣言されます。関数は, 「型の配列」または「型を返す関数」以外のどのような型の値でも返すことができます。配列と関数へのポインタを返すことができます。返却値がある場合には, `return` 文の式に指定します。 `return` 文を実行すると関数の実行が終了し, 呼出し関数に制御が返されます。値を返す関数では, 関数の返す型と互換性のある型を持つ式を次に示す書式で `return` の後に続けることができます。

```
return 式;
```

必要に応じて, 式は関数の返す型へ変換されます。関数が返す値は左辺値ではないことに注意してください。したがって, 関数呼出しは代入演算子の左側に置くことはできません。

次に, 文字を返す関数定義の例を示します。

```
char letter(char param1)
{
    .
    .
    .
    return param1;
}
```

呼出し関数は返却値を無視できます。returnの後に式が指定されない場合、または右中括弧によって関数が終了する場合の返却値は定義されていません。void関数の場合には値は返されません。

関数が値を返さない場合、または関数が値を必要としない状況から常に呼び出される場合には、返却値の型として void を指定する必要があります。

```
void message()  
{  
    printf("This function has no return value.");  
    return;  
}
```

以下の場合に関数定義または関数宣言において返却値の void型を指定すると、エラーが発生します。

- 関数が値を返そうとすると、return文でエラーが生じます。
- 値を必要とする状況で void関数が呼び出されると、関数呼出し側でエラーが生じます。

宣言子

宣言子は宣言される関数名を指定します。次の例の f1 に示すように、宣言子は単一識別子のように単純な形式で指定できます。

```
int f1(char p2)
```

次の例では、f1 は「int を返す関数」です。また、次の例のように宣言子を複雑な構文にすることもできます。

```
int (*(f1(int x))[5])(float)
```

この例では、f1は「int を返す (float実引数をとる) 関数を指示する 5 つのポインタの配列に、さらにポインタを返す (int実引数をとる) 関数」です。特定の宣言子の構文については、第 4 章を参照してください。

宣言子 (関数) はあらかじめ宣言しておく必要はありません。関数があらかじめ宣言された場合には、その関数定義の仮引数型と返却値の型は前の関数宣言と等しくなければなりません。

関数

5.3 関数定義

宣言子には、関数の仮引数並びを含めることができます。 *Compaq C* では、最大で 253 個の仮引数をコンマで区切って表した並びを括弧で囲んで指定することができます。各仮引数は特に指定しない限り auto 記憶域クラスになりますが、register も使用できます。仮引数並びの右括弧の後にセミコロンは付けません。

関数の仮引数を指定する方法には、次の 2 種類があります。

- 新しい形式、すなわちプロトタイプ形式

これは仮引数型並びを含んでいます。次にその例を示します。

```
int f1(char a, int b)
{
    function body
}
```

- 旧形式

これは識別子並びを含んでいます。仮引数型は関数定義内の別の宣言並びに定義します。この宣言並びは関数本体を開始する左中括弧の前に指定します。次にその例を示します。

```
int f1(a, b)
char a;
int b;
{
    function body
}
```

宣言されない仮引数は int 型とみなされます。

仮引数を持たない関数定義は空の仮引数並びで定義されます。この空の仮引数並びは、次の 2 つの方法のいずれかで指定されます。

- プロトタイプ形式を使用する場合には void キーワードを使用します。次にその例を示します。

```
char msg(void)
{
    return 'a';
}
```

- 旧形式を使用する場合には空の括弧を使用します。次にその例を示します。

```
char msg()  
{  
    return 'a';  
}
```

プロトタイプ形式を使用して定義される関数は、その関数のプロトタイプを設定します。プロトタイプは、同じ関数の先行する宣言または後に続く宣言と一致しなければなりません。

旧形式を使用して定義される関数は、プロトタイプを設定しません。しかし、その関数の以前の宣言によるプロトタイプが存在する場合には、省略時の仮引数が定義の仮引数として適用され、その定義の仮引数宣言がプロトタイプの各仮引数宣言と厳密に一致していなければなりません。

1つの関数に、旧形式とプロトタイプ形式の宣言と定義を混用しないでください。使用することはできますが推奨できません。

関数の仮引数と実引数については第 5.6 節を、関数プロトタイプについては第 5.5 節を参照してください。

複文

複文は、関数本体またはループ本体の中括弧で囲まれた一連の宣言と文からなります。関数本体は左中括弧 ({) で始まり、右中括弧 (}) で終了します。この 2 つの中括弧の間には有効な C の宣言と文が入ります。1 つ以上の return 文を含めることはできますが、必須ではありません。

5.4 関数宣言

関数の返却値が int の場合には、関数を宣言せずに呼び出すことができます。ただし、この方法では型検査が行われないため推奨できません。すべての関数を宣言してください。返却値が別の型の場合で、関数定義がソース・コード中の呼出し関数の後に指定される場合には、その関数はそれを呼び出す前に宣言する必要があります。次にその例を示します。

関数

5.4 関数宣言

```
char lower(int c);                /* Function declaration */
caller()                          /* Calling function   */
{
    int c;
    char c_out;
    .
    .
    .
    c_out = lower(c);            /* Function call      */
}
char lower(int c_up)              /* Function definition */
{
    .
    .
    .
}
```

ソース・コード中で caller関数の前に lowerの関数定義が位置している場合には、lowerを呼び出す前に再度宣言する必要はありません。この場合、関数定義はそれ自体の宣言として機能し、同じソース・ファイル中でその後に定義されるすべての関数からの関数呼出しのスコープに入っています。

lowerの関数定義と関数宣言はいずれも、プロトタイプ形式であることに注意してください。Compaq Cでは、仮引数型を関数宣言子に指定しない旧形式の関数宣言をサポートします。ただし、効率のよいプログラミング方法としては、プログラム中のすべてのユーザ定義関数にプロトタイプ宣言を使用して、関数を使用する前に各プロトタイプを指定します。さらに、関数宣言の仮引数識別子は、関数定義の仮引数識別子とは異なっても有効であることに注意してください。

関数宣言では、voidキーワードを使用して、空の実引数並びを指定しなければなりません。次にその例を示します。

```
char function_name(void);
```

関数定義では、voidキーワードを関数宣言に使用して、値を返さない関数の返却値の型を指定することもできます。次にその例を示します。


```
main()
{
    void function_name( );
    .
    .
    .
}
void function_name( )
{ }
```

5.5 関数プロトタイプ

関数プロトタイプは、仮引数並びに実引数のデータ型を指定する関数宣言です。コンパイラは関数プロトタイプの情報を使用して、プロトタイプのスコープ内の対応する関数定義、および対応するすべての関数宣言と関数呼出しに正しい数の実引数や仮引数が含まれていることを確認し、その各実引数や仮引数が正しいデータ型のものであることを確認します。

プロトタイプは、旧形式の関数宣言とは構文的に区別されます。これら 2 つの形式は単一の関数で混用できますが、推奨できません。次は、旧形式の宣言とプロトタイプ形式の宣言を比較したものです。

旧形式:

- 関数は呼出し時に暗黙に宣言できます。
- 関数の実引数は、呼出しの前に省略時の値に変換されます。
- 実引数の数と型は検査されません。

注意

*Compaq C*コンパイラの旧形式の関数宣言が厳密な ANSI 規格モードの場合、またはコンパイラの検査オプションが指定されている場合にのみ警告を出します。

プロトタイプ形式:

- 関数は、呼び出す前にプロトタイプで明示的に宣言します。複数の宣言には互換性がなければなりません。各仮引数型は正確に一致しなければなりません。
- 関数の実引数は仮引数の宣言された型へ変換されます。
- 実引数の数と型はプロトタイプで宣言したものと照合され、宣言した型と一致するかその型に変換できるものでなければなりません。空の仮引数並びは `void` キーワードを使用して指定します。
- プロトタイプの仮引数並びに省略記号を使用すると、可変個の仮引数があることを示します。

5.5.1 プロトタイプの構文

関数プロトタイプの構文は、次のとおりです。

関数プロトタイプ宣言:

宣言指定子_{opt} 宣言子;

宣言子は仮引数型並びを含んでおり、この並びがその関数の仮引数の型の指定と識別子の宣言を行います。

仮引数型並びを `void` 型の単一仮引数で構成すると、関数には仮引数がないという指定になります。

仮引数型並びは、`[*]` という表記で指定した可変長配列であるメンバを含むことができます。

最も簡単な形式では、関数プロトタイプ宣言は次の書式になります。

記憶域クラス_{opt} 返す型_{opt} 関数名 (型₁ 仮引数₁ , ... , 型_n 仮引数_n);

次の関数定義を例にとります。

```
char function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

次は、この関数に対応するプロトタイプ宣言です。

```
char function_name( int lower, int *upper, char (*func)(), double y );
```

プロトタイプはプロトタイプ形式に指定した対応する関数定義のヘッダと同一であり、必要に応じて終了のセミコロン(;)またはコンマ(,)を付けます。それは、そのプロトタイプが単独で宣言されているか、複数宣言中のものであるかによって異なります。

関数プロトタイプは、それに対応する関数定義と同じ仮引数識別子を使用する必要はありません。これは、プロトタイプの識別子が識別子並び内のみのスコープを持つためです。さらに、識別子自体はプロトタイプ宣言に指定する必要はなく、その型だけが必要です。

たとえば、次のプロトタイプ宣言はすべて等しいものです。

```
char function_name( int lower, int *upper, char (*func)(), double y );
char function_name( int a, int *b, char (*c)(), double d );
char function_name( int, int *, char (*)(), double );
```

プログラムの明確さを向上し、コンパイラの型検査能力を高めるためには、できるだけ識別子をプロトタイプに含めるようにしてください。

可変個の実引数並びは、関数プロトタイプに省略記号を含めることによって指定します。最低 1 つの仮引数を、省略記号の前に指定しなければなりません。次にその例を示します。

```
char function_name( int lower, ... );
```

データ型の指定は、関数プロトタイプから省略することはできません。

C99 規格では、関数のプロトタイプ宣言での仮パラメータの最も外側の配列境界で、static キーワードを使用できます。このキーワードを使用すると、関数の呼び出しごとに、少なくとも配列境界の宣言で指定した配列要素と同じ数だけ、対応する実引数がアクセス可能であることをコンパイラに知らせることができます。たとえば、次の 2 つの関数定義があるとします。

関数

5.5 関数プロトタイプ

```
void foo(int a[1000]){ ... }  
void bar(int b[static 1000]) { ...}
```

このfooの宣言は、aをint *として宣言するのとまったく同じです。fooの本体をコンパイルするとき、配列要素がいくつあるかについては、コンパイラは何もわかりません。barの宣言は、少なくとも 1000 個の配列要素が存在し、安全にアクセス可能であることをコンパイラが前提とできる点が異なります。この目的は、安全にプリフェッチできる配列要素の範囲をオプティマイザに知らせることです。

5.5.2 スコープと変換

プロトタイプは、プログラムの各コンパイル単位に適切に配置しなければなりません。プロトタイプの位置は、そのスコープを決定します。関数プロトタイプがそれに対応する関数呼出しのスコープ内とみなされるのは他の関数宣言の場合と同様に、プロトタイプが関数呼出しと同じブロック内で指定される場合、ネストされたブロック内で指定される場合、またはソース・ファイルの一番外側のレベルで指定される場合のみです。コンパイラは、プロトタイプの位置からそのスコープの終わりまですべての関数定義、宣言、および呼出しを検査します。プロトタイプの位置を間違えて配置し、関数定義、宣言、または呼出しがプロトタイプのスコープ外で行われた場合には、その関数呼出しはプロトタイプがないものとして処理されます。

関数プロトタイプの構文は各関数定義の関数ヘッダを抽出し、セミコロン(;)を追加し、ヘッダに各プロトタイプを配置し、プログラム中の各コンパイル単位の先頭にそのヘッダを含めることを可能にするために設計されています。この方法によって関数プロトタイプは外部宣言され、そのプロトタイプのスコープはコンパイル単位全体に拡張されます。*Compaq C*ライブラリ関数呼出しのプロトタイプ検査を使用するには、プログラムが使用するライブラリ関数を含む .hファイルを呼び出すための #include前処理命令を指定します。

関数定義、宣言、または呼出しの各実引数の数がプロトタイプと一致しない場合には、エラーが発生します。

関数呼出しの実引数のデータ型が関数プロトタイプの対応する型と一致しない場合、コンパイラは変換を試みます。一致しない実引数がプロトタイプ仮引数と代入互換性がある場合、コンパイラはその実引数を実引数の変換の規則 (第 5.6.1 項を参照) によって、プロトタイプに指定されているデータ型に変換します。

不一致の実引数とプロトタイプ仮引数の間に代入互換性がない場合は、エラー・メッセージが出されます。

5.6 仮引数と実引数

C 関数は、仮引数と実引数を通じて情報を交換します。仮引数とは、関数宣言または定義において関数名の後に続く括弧内の宣言のことです。実引数とは関数呼出しの括弧内の式のことです。

次に、C 関数の仮引数と実引数に適用される規則を示します。

- 可変個の実引数並びを持つ関数以外は、関数呼出しの実引数の数は関数定義の仮引数の数と同じでなければならない。この数は 0 にすることもできる。
- 実引数 (とそれに対応する仮引数) の最大数は、単一関数において 253 個ある。
- 実引数はコンマで区切られる。ただし、コンマはこの文脈の演算子ではないため、コンパイラは各実引数を任意の順序で評価できる。ただし、実際の呼出しの前には評価順序点がある。
- 実引数は値によって渡される。つまり、関数が呼び出されると仮引数はその実引数のアドレスではなく、その値のコピーを受け取る。この規則は、実引数として渡されるスカラ値、構造体、共用体のすべてに適用される。
- 仮引数を変更しても、関数呼出しによって渡されたそれに対応する実引数は変更されない。ただし、実引数にアドレスまたはポインタを指定できるため、関数はそのアドレスを使用して、呼び出した関数の中に定義されている変数の値を変更できる。
- 旧形式では、明示的に宣言されない仮引数には省略時の `int` 型が割り当てられる。

- 関数の仮引数のスコープはその関数内である。したがって、異なる関数に同じ名前の仮引数があっても関連はない。

5.6.1 実引数の変換

関数呼出しでは、評価される実引数の型はそれに対応する仮引数の型と一致しなければなりません。一致しない場合には、次の変換が行われます。この方法は、プロトタイプが関数のスコープ内にあるかどうかによって異なります。

- プロトタイプ形式で指定した関数の実引数は、そのプロトタイプに指定されている仮引数型に変換されます。ただし、省略記号 (...) に対応する実引数が、プロトタイプがスコープにない場合と同様に変換される場合は除きます (この場合は、もう一方の規則が適用されます)。次にその例を示します。

```
void f(char, short, float, ...);
```

```
char c1, c2;
```

```
short s1,s2;
```

```
float f1,f2;
```

```
f(c1, s1, f1, c2, s2, f2);
```

c1, s1, および f1 実引数はそれぞれの型で渡されます。これに対して, c2, s2, f2 実引数は, それぞれ int, int, double に変換されます。

- スコープにプロトタイプを持たない関数の実引数は、仮引数の型へ変換されません。その代わり、実引数並び内の式が次の規則に従って変換されます。
 - float 型の実引数は double に変換される。
 - char, unsigned char, short, および unsigned short 型の実引数は int に変換される。
 - コモン C の互換性モードでのコンパイル時には, *Compaq C* は unsigned char または unsigned short 型の実引数を unsigned int 型に変換する。

実引数では、これ以外の省略時の変換は行われません。特定の実引数をそれに対応する仮引数の型と一致させるために変換が必要な場合には、キャスト演算子を使用します。キャスト演算子については、第 6.4.6 項を参照してください。

5.6.2 実引数としての関数識別子と配列識別子

関数識別子と配列識別子は、関数の実引数として指定できます。関数識別子は括弧なしで指定し、配列識別子は中括弧なしで指定します。関数識別子または配列識別子が指定されると、その関数または配列のアドレスとして評価されます。さらに、その返却値が整数であっても、その関数を宣言または定義する必要があります。例 5-1 では、実引数として渡す関数の宣言の方法と時期、および渡し方を示しています。

例 5-1 実引数として渡される関数の宣言

```

1 int x() { return 25; }           /* Function definition and */
   int z[10];                     /* array defined before use */

2 fn(int f1(), int (*f2)(), int a1[]) /* Function definition */
   {
       f1();                      /* Call to function f1 */
       .
       .
       .
   }

   void caller(void)
   {
3       int y();                  /* Function declaration */
       .
       .
       .
4       fn(x, y, z);              /* Function call: functions */
                                   /* x and y, and array z */
                                   /* passed as addresses */
       .
       .
       .
   }
   int y(void) { return 30; }      /* Function definition */

```

次は、例 5-1 の説明です。

- 1 x関数は別の場所で宣言することなく、実引数並びに渡すことができます。これは、caller関数の前にある定義がその宣言として機能するからです。

関数

5.6 仮引数と実引数

- 2 関数を表す仮引数は、関数または関数へのポインタのいずれかとして宣言できます。配列を表す仮引数は、配列または配列要素の型へのポインタのいずれかとして宣言できます。次にその例を示します。

```
fn(int f1(), int f2(), int a1[]) /* f1, f2 declared as */
{...} /* functions; a1 declared */
/* as array of int. */

fn(int (*f1)(), int (*f2)(), int *a1) /* f1, f2 declared as */
{...} /* pointers to functions; */
/* a1 declared as pointer */
/* to int. */
```

このような仮引数が関数または配列として宣言されると、コンパイラはそれに対応する実引数を自動的にポインタへ変換します。

- 3 関数定義が caller 関数の後に位置するため、y 関数は実引数並びに渡される前に宣言しなければなりません。
- 4 実引数として関数を渡す場合は括弧を含めてはなりません。同様に、配列を指定する場合も添字を含めてはなりません。

5.6.3 main 関数への実引数の引渡し

プログラムの起動時に呼び出される関数を `main` と呼びます。main 関数は仮引数なしで、または 2 つの仮引数を指定して定義できます。この 2 つの仮引数は、main 関数の実行開始時にコマンド行実引数をプログラムに渡すためのものです。2 つの仮引数は、ここでは `argc` および `argv` という名前で説明しますが、それらを宣言する関数の中でのみ有効であるため任意の名前を使用することもできます。main 関数の構文は次のとおりです。

```
int main { ... }

int main( int argc, char *argv[ ] ){ ... }
```

`argc`

プログラムを起動したコマンド行にある実引数の数であり、負以外の値で表されます。

`argv`

実引数を含む文字列の配列へのポインタであり、各文字列ごとに1つずつ指定されます。`argv[argc]`値は空ポインタです。

`argc` の値が0よりも大きい場合、配列メンバの `argv[0]` から `argv[argc - 1]` までは文字列へのポインタを含みます。これには、プログラムの起動前のホスト環境による処理系定義値が与えられています。この目的は、ホスト環境以外の場所からプログラムの起動が行われる前に判別された情報をそのプログラムに提供することにあります。たとえば、ホスト環境が大小文字の両方を含む文字列を提供できない場合には、そのホスト環境は文字列を小文字で受け取ることを保証します。

`argc` の値が0よりも大きい場合、`argv[0]` で示される文字列はプログラム名を表します。プログラム名がホスト環境から使用できない場合、`argv[0]` はヌル文字です。`argc`値が1よりも大きい場合、`argv[1]` から `argv[argc - 1]` で示される文字列はプログラムの仮引数を表します。

`argc` および `argv` 仮引数と `argv` 配列で示される文字列は、プログラムによって変更できます。最後に格納された値は、プログラムの起動から終了までの間だけ保持されます。

`main` 関数定義における仮引数はオプションです。ただし、アクセスできる仮引数は定義した仮引数のみです。

`main` 関数との実引数の引渡しについては、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

式とは *Compaq C* の一連の演算子とオペランドからなり、値を生成するか、または副作用を生成するものです。最も単純な式は定数と変数名から構成され、値は直接生成されます。その他の式は、演算子と部分式を組み合わせて値を生成します。式は値と型の両方を持ちます。

特に指定しない限りは部分式の評価順序および副作用の発生順序はありません。したがって、順序に依存するコーディングを行うと、予測できない結果となる可能性があります。

式の各オペランドは互換型を持たなければなりません。場合によっては、オペランドのデータ型に互換性を持たせるためにコンパイラが変換を行います。

この章では式と演算子について説明します。

- 1 次式と演算子 (第 6.1 節)
- C 言語演算子 (第 6.2 節)
- 後置演算子 (第 6.3 節)
- 単項式と演算子 (第 6.4 節)
- 2 項式と演算子 (第 6.5 節)
- 条件式と演算子 (第 6.6 節)
- 代入式と演算子 (第 6.7 節)
- コンマ式と演算子 (第 6.8 節)
- 定数式 (第 6.9 節)
- 複合リテラル式 (第 6.10 節)
- データ型の変換 (第 6.11 節)

6.1 1 次式

単純な式は 1 次式と呼ばれ値を示します。1 次式には、あらかじめ宣言された識別子、定数、文字列リテラル、および括弧で囲んだ式が含まれます。

1 次式の構文は次のとおりです。

1 次式

識別子
定数
文字列リテラル
(式)

これ以降の各項では 1 次式について説明します。

6.1.1 識別子

識別子は、オブジェクトまたは関数を指定するときに宣言される 1 次式です。

型が算術演算、構造体、共用体、またはポインタの場合、オブジェクトを指定する識別子は左辺値です。配列名はその配列の最初の要素のアドレスに評価されます。配列名は左辺値ですが、可変左辺値ではありません。

関数を指定する識別子は関数指名子と呼ばれます。関数指名子は関数のアドレスに評価されます。

6.1.2 定数

定数は 1 次式であり、その型は形式 (整数型、文字型、浮動小数型、または列挙型) によって異なります (第 1.9 節を参照してください)。定数を左辺値にすることはできません。

6.1.3 文字列リテラル

文字列リテラルは 1 次式であり，その型は形式 (文字型または wchar_t 型) によって異なります (第 1.9 節を参照してください)。文字列リテラルは左辺値です。

6.1.4 括弧で囲んだ式

括弧内の式が持つ型と値は，括弧のない式が持つ型と値と同じです。式を括弧で区切ることによって，演算子の優先順位を変更することができます。

6.2 C 言語演算子

変数と定数は，C 言語演算子と結合することでより複雑な式を作成することができます。表 6-1 は C 言語演算子の一覧です。

表 6-1 C 言語演算子

演算子	例	説明
()	f()	関数呼出し
[]	a[10]	配列の参照
->	s->a	構造体および共用体メンバの選択
.	s.a	構造体および共用体メンバの選択
+ [単項]	+a	a の値
- [単項]	-a	a の負数
* [単項]	*a	アドレス a のオブジェクトの参照
& [単項]	&a	a のアドレス
~	~a	a の 1 の補数
++ [前置]	++a	増分後の a の値
++ [後置]	a++	増分前の a の値
-- [前置]	--a	減分後の a の値

(次ページに続く)

表 6-1 (続き) C 言語演算子

演算子	例	説明
-- [後置]	a--	減分前の a の値
sizeof	sizeof (t1)	t1 型を持つオブジェクトのサイズのバイト数
sizeof	sizeof e	式 e の型を持つオブジェクトのサイズのバイト数
__typeof__	__typeof__ (t1)	型 t1 の型
__typeof__	__typeof__ (e)	式 e の型
_Pragma	_Pragma (string-literal)	string-literal の非文字列化
+ [2 項]	a + b	a と b の和
- [2 項]	a - b	a 引く b の差
* [2 項]	a * b	a と b の積
/	a / b	a 割る b の商
%	a % b	a 割る b の余り
>>	a >> b	a を b ビットだけ右にシフトする
<<	a << b	a を b ビットだけ左にシフトする
<	a < b	a < b の場合 1, それ以外は 0
>	a > b	a > b の場合 1, それ以外は 0
<=	a <= b	a <= b の場合 1, それ以外は 0
>=	a >= b	a >= b の場合 1, それ以外は 0
==	a == b	a が b に等しい場合 1, それ以外は 0
!=	a != b	a が b と等しくない場合 1, それ以外は 0
& [2 項]	a & b	a と b のビット AND
	a b	a と b のビット OR
^	a ^ b	a と b のビット XOR (排他的 OR)
&&	a && b	a と b の論理積 (0 または 1 になる)
	a b	a と b の論理和 (0 または 1 になる)
!	!a	a の否定 (0 または 1 になる)
?:	a ? e1 : e2	a が 0 以外の場合式 e1 a が 0 の場合式 e2

(次ページに続く)

表 6-1 (続き) C 言語演算子

演算子	例	説明
=	a = b	b を a に代入
+=	a += b	a と b の和 (a に代入)
-=	a -= b	a 引く b の差 (a に代入)
*=	a *= b	a と b の積 (a に代入)
/=	a /= b	a 割る b の商 (a に代入)
%=	a %= b	a 割る b の余り (a に代入)
>>=	a >>= b	a を b ビット右にシフトする (a に代入)
<<=	a <<= b	a を b ビット左にシフトする (a に代入)
&=	a &= b	a AND b (a に代入)
=	a = b	a OR b (a に代入)
^=	a ^= b	a XOR b (a に代入)
,	e1, e2	e2 (e1 が最初に評価される)

C 演算子は、次のように分類されます。

- 単一オペランドに続く後置演算子
- 単一オペランドの前にくる単項前置演算子
- 2つのオペランドをとり、さまざまな算術演算と論理演算を行う 2 項演算子
- 3つのオペランドをとり、最初の式の評価により 2 番目または 3 番目の式を評価する条件演算子 (3 項演算子)
- 値を変数に代入する代入演算子
- コンマで区切った式を左から右に評価するコンマ演算子

演算子の優先順位は、1つの式の中の項のグループ分けを決定します。優先順位は、式を評価する方法に影響を与えます。演算子の中には、他の演算子よりも優先順位の高いものがあります。たとえば、乗算演算子は加算演算子よりも優先順位が高くなります。

```
x = 7 + 3 * 2;          /* x is assigned 13, not 20 */
```

この文は、次の文に等しくなります。

```
x = 7 + ( 3 * 2 );
```

式と演算子

6.2 C 言語演算子

式に括弧を使用すると、省略時の優先順位が変わります。次にその例を示します。

```
x = (7 + 3) * 2; /* (7 + 3) is evaluated first */
```

括弧で囲まれていない式では、高い優先順位を持つ演算子は低い優先順位を持つ演算子よりも前に評価されます。次にその例を示します。

```
A + B * C
```

乗算演算子 (`*`) は加算演算子 (`+`) よりも優先順位が高いため、`B` と `C` の識別子の乗算が最初に行われます。

表 6-2 は、C 言語演算子を評価するためにコンパイラが使用する優先順位を示しています。最も高い優先順位の演算子を表の先頭に示し、最も低い優先順位の演算子を表の最後に示します。

表 6-2 C 言語演算子の優先順位

分 類	演 算 子	結 合 規 則
後置 () [] - > . ++ --	左から右	
単項	+ - ! ~ ++ -- (型) * & sizeof	右から左
乗除	* / %	左から右
加減	+ -	左から右
シフト	<< >>	左から右
関係	< <= > >=	左から右
等価	= = ! =	左から右
ビット AND	&	左から右
ビット XOR	^	左から右
ビット OR		左から右
論理積	&&	左から右

(次ページに続く)

表 6-2 (続き) C 言語演算子の優先順位

分 類	演 算 子	結 合 規 則
論理和		左から右
条件	?:	右から左
代入	= += -= *= /= %= >>= <<= &= ^= =	右から左
コンマ	,	左から右

結合規則は優先順位と関連しているため、同じ優先順位を持つ演算子のグループ分けの際の不明瞭さが解消されます。次の文は C 言語の規則により `a * b` が最初に評価されます。

```
y = a * b / c;
```

より複雑な式の場合、たとえば次の例では結合規則は `b ? c : d` を最初に評価するように指定します。

```
a ? b ? c : d : e;
```

条件演算子の結合規則は、行の右から左に評価されます。また、代入演算子も右から左に評価されます。次にその例を示します。

```
int x = 0 , y = 5, z = 3;  
x = y = z; /* x has the value 3, not 5 */
```

その他の演算子は左から右に評価されます。たとえば、2 項の加算、減算、乗算、除算演算子はすべて左から右の結合規則を持ちます。

結合規則は表 6-2 に示した演算子の各列に適用され、右から左に評価されるものと左から右に評価されるものがあります。括弧で囲まれていない式では、結合規則により同じ列にある各演算子の評価順序が決まります。次にその例を示します。

```
A * B % C
```

この式は乗除演算子 (* , / , %) が左から右に評価されるため、次のように評価されます。

(A * B) % C

式の中の優先順位と結合規則を制御するために、常に括弧を使用できます。

6.3 後置演算子

後置式には、配列の参照、関数呼出し、構造体参照または共用体参照、後置インクリメント式と後置デクリメント式が含まれます。後置式は、左から右への結合規則を持ちます。

後置式の構文は、次のとおりです。

後置式:

- 配列の参照
- 関数呼出し
- 構造体および共用体メンバの参照
- 後置インクリメント式
- 後置デクリメント式

6.3.1 配列の参照

大括弧演算子 ([]) は、配列要素を参照するために使用します。配列の参照の構文は、次のとおりです。

配列の参照:

後置式 [式]

たとえば 1 次元配列の場合、配列内の特定要素は次に示すように参照することができます。

```
int sample_array[10]; /* Array declaration; array has 10 elements */
sample_array[0] = 180; /* Assign value to first array element      */
```

この例は、配列の最初の要素である `sample_array[0]` に 180 の値を代入しています。C 言語では、0 を起点とする配列の添字付けを使用することに注意してください。

2 次元配列 (厳密には、配列の配列) では、次に示すように配列内の特定要素を参照できます。

```
int sample_array[10][5]; /* Array declaration; array has 50 elements */
sample_array[9][4] = 180; /* Assign value to last array element */
```

この例は、要素である `sample_array[9][4]` に 180 の値を代入しています。

概念上からは、多次元配列は配列の配列の配列の....という型になります。したがって、配列の参照が完全に修飾されていない場合には、指定されていない次元の最初の要素のアドレスを参照することになります。次にその例を示します。

```
int sample_array[10][5]; /* Array declaration */
int *p1; /* Pointer declaration */
p1 = sample_array[7]; /* Assigns address of subarray to pointer */
```

この例では、`p1` には 1 次元の部分配列である `sample_array[7]` にある最初の要素のアドレスが入っています。この例のように、部分的に修飾した配列を右辺値として使用することはできますが、左辺値としては完全に修飾した配列の参照しか使用できません。たとえば、*Compaq C* では次のような文は使用できません。これは、この文が配列の 2 次元を省略しているからです。

```
int sample_array[10][5]; /* Array declaration */
sample_array[7] = 21; /* Error */
```

大括弧のない配列名の参照を使用して、次に示す文のように配列アドレスを関数に渡すことができます。

```
func(sample_array);
```

また、大括弧演算子を使用して、次に示すように通常のポインタ算術演算を実行することもできます。

```
p1[intexp]
```

式と演算子

6.3 後置演算子

この場合、`p1` はポインタであり、`intexp` は整数値の式です。式の結果は `p1` が示す値になります。この値は、アドレス指定したオブジェクト (配列要素) のバイト単位のサイズを `intexp` の値に掛けて増分された値です。式 `* (p1 + intexp)` と式 `p1[intexp]` は、等しい式として定義されます。両方の式とも、同じメモリ位置を参照し同じ型を持ちます。配列の添字付けは交換可能です。すなわち、`intexp[p1]` は `p1[intexp]` に等しいものです。添字付きの式は常に左辺値になります。

6.3.2 関数呼出し

関数呼出しの構文は、次のとおりです。

関数呼出し:

後置式

実引数式並び_{opt}:

代入式

代入式並び_{opt} , 代入式

関数呼出しは関数指名子からなる後置式であり、後に括弧が続きます。関数の仮引数並びの式の評価順序は定義されていませんが、実際の呼出しの前には評価順序点が存在します。括弧内には実引数並びを (コンマで区切って) 含めることも、括弧内を空にすることもできます。呼び出される関数が宣言されていなければ、`int` を返す関数とみなされます。

配列または関数である実引数を渡すには、識別子を大括弧または括弧を付けずに実引数並びに指定してください。コンパイラは、その配列または関数のアドレスを呼び出されるルーチンに渡します。つまり、呼び出される関数に対応する仮引数はポインタとして宣言しなければなりません。

次の例では、`func1` は `double` を返す関数として宣言されています。仮引数数と型は指定されていません。

```
double func1();
```

その後、func1関数は次に示すように関数呼出しに使用できます。

```
result = func1(c);  
      または  
result = func1();
```

また、func1識別子は括弧を付けずに他の文脈で使用できます。次に、別の関数呼出しの実引数として使用している例を示します。

```
dispatch(func1);
```

この例では、func1関数のアドレスが dispatch関数に渡されます。識別子が「... を返す関数」型として宣言された場合、その識別子が括弧のない実引数として渡されるときに通常、「... を返す関数のアドレス」に変換されます。唯一の例外は、関数指名子が単項&演算子のオペランドである場合です。この場合、変換は明示的に行われます。

また、関数へのポインタを逆参照することにより関数を呼び出すこともできます。次の例では、pf は double を返す関数へのポインタとして宣言され、func1関数のアドレスが代入されます。

```
double (*pf)( );  
      .  
      .  
      .  
pf = func1;
```

func1関数は、次に示すように呼び出すことができます。

```
result = (*pf)();
```

この関数呼出しは有効ですが、次に示す形式の方が簡潔です。

```
result = pf();
```

関数呼出しにおいて呼び出される関数を示す式の型がプロトタイプを含んでいない場合、第 6.11.3 項で説明する汎整数拡張がそれぞれの適用可能な実引数で実行され、float型を持つ実引数は double に変換されます。これらを省略時の実引数の拡張と呼びます。渡される実引数の数が仮引数の数と一致しない場合の動作結果は

定義されていません。また、関数がプロトタイプを含まない型で定義され、さらに拡張後の実引数の型が拡張後の仮引数型と互換性がない場合の動作結果も定義されていません。関数がプロトタイプを含む型で定義され、さらに拡張後の実引数の型が仮引数型と互換性がないか、またはプロトタイプが(可変個の仮引数並びを示す)省略記号で終了した場合も同様に動作結果は定義されていません。

呼び出される関数を示す式がプロトタイプを含む型を持っている場合には、渡される実引数はそれに対応する仮引数の型に暗黙に変換されます。関数プロトタイプに省略記号が含まれていると、最後に宣言した仮引数の後で実引数型変換が停止します。実引数が残っている場合には、省略時の実引数の拡張が実行されます。呼び出される関数を示す式が示す型と互換性のない型で関数が定義された場合の動作結果は定義されていません。

その他の変換は、暗黙には行われません。特に実引数の数および型が、プロトタイプを含まない関数定義の仮引数の数および型と比較されることはありません。

再帰的関数呼出しは、直接的にも間接的にも他の関数の連鎖を通して許可されています。

6.3.3 構造体と共用体の参照

構造体または共用体メンバは、ドット演算子(.)を使用して直接的に、または矢印演算子(->)を使用して間接的に参照することができます。

構造体および共用体の参照(つまり構成要素の選択)の構文は、次のとおりです。

構造体および共用体の参照:

後置式 . 識別子
後置式 -> 識別子

矢印演算子は常に左辺値を生成します。ドット演算子は、後置式が左辺値である場合に左辺値を生成します。

直接メンバ選択では、最初のオペランドは構造体または共用体を指定しなければなりません。さらに、識別子はその構造体または共用体の宣言されたメンバを指名しなければなりません。

間接メンバ選択では、最初のオペランドは構造体または共用体のポインタでなければなりません。さらに、識別子はその構造体または共用体の宣言されたメンバを指名しなければなりません。矢印演算子はハイフン (-) と右不等号 (>) で指定され、構造体または共用体メンバの参照を指定します。式 $E1 \rightarrow name$ は、定義上では厳密に $(*E1).name$ と同じです。また、これは $E2$ が左辺値である場合、 $E2.name$ が $(&E2) \rightarrow name$ と同じであることも意味しています。

指名された構造体メンバは、完全に修飾しなければなりません。すなわち、そのメンバの前にピリオド、矢印、またはその両方で区切られた上位レベル・メンバの一連の名前がなくてはなりません。式の値は構造体または共用体の指名されたメンバであり、その型がそのメンバの型になります。構造体と共用体については、第 3.4.4 項と第 3.4.5 項を参照してください。

1 つの例外を除き、値を共用体の 1 つのメンバに格納した後でその共用体の異なるメンバにアクセスすると、その結果は参照したメンバのデータ型と共用体内の境界調整によって異なります。

この例外とは、共用体の使用を簡単にするためのものです。共用体に共通の初期シーケンスを共有する複数の構造体が含まれるとき、共用体がこれらの構造体の 1 つを現在含んでいる場合には、それらの共通の初期部分を調べることができます。2 つの構造体に対応するメンバが 1 つまたは複数の初期メンバのシーケンスの互換型 (ビット・フィールドでは同じ幅) を持つ場合には、その 2 つの構造体は共通の初期シーケンスを共有します。

6.3.4 後置インクリメント演算子と後置デクリメント演算子

C 言語には、スカラ型のオブジェクトの増分や減分を行う 2 つの単項演算子があります。後置インクリメントの構文は、次のとおりです。

後置インクリメント式:

後置式 ++

式と演算子

6.3 後置演算子

後置デクリメントの構文は、次のとおりです。

後置デクリメント式:

後置式 --

オペランドがポインタである場合を除き、インクリメント演算子である ++ はそのオペランドに 1 を加算し、デクリメント演算子である -- はそのオペランドから 1 を減算します。オペランドが T へのポインタ型のポインタである場合、そのポインタは、sizeof で増分 (または減分) されます。これにより、T 型のオブジェクトの配列内の次の (または前の) 要素が示されます。

++ と -- の両方を前置演算子 (オペランドの前、つまり ++n)、または後置演算子 (オペランドの後、つまり n++) として使用できます。これら両方の場合とも n を増分します。式 ++n はその値が使用される前に n を増分し、n++ はその値が使用された後に n を増分します。

インクリメント演算子とデクリメント演算子の前置形式については、第 6.4.3 項で説明します。この項では後置形式について説明します。

次の式を例にとります。

左辺値 ++

後置演算子 ++ は、定数 1 をオペランドに加えてそのオペランドを変更します。式の値は 1 だけ増分されたオペランドの値、あるいは増分される前のオペランドの古い値です。次にその例を示します。

```
int i, j;
j = 5;
j++;           /* j = 6 (j incremented by 1) */
i = j++;       /* i = 6, j = 7                */
```

インクリメント演算子とデクリメント演算子を使用する場合には、式の評価の順序に依存しないようにしてください。次の不明瞭な式を例にとります。

```
k = x[j] + j++;
```


`x[j]` の `j` 値は `j` の増分前に評価されるのか、または増分後に評価されるのかが指定されていません。不明瞭さを避けるために、変数の増分を別の文で行います。次にその例を示します。

```
j++;  
k = x[j] + j;
```

`++` 演算子と `--` 演算子は、浮動小数点オブジェクトとともに使用することもできます。この場合には、1.0 ずつオブジェクトをスケーリングします。

6.4 単項演算子

単項式は、単項演算子を単一オペランドと組み合わせることによって形成されます。すべての単項演算子の優先順位は等しく、右から左への結合規則を持ちます。単項演算子には次の種類があります。

- 単項マイナス演算子 (`-`) と単項プラス演算子 (`+`) (第 6.4.1 項を参照のこと)
- 否定 (`!`) (第 6.4.2 項を参照のこと)
- 前置インクリメント (`++`) と前置デクリメント (`--`) (第 6.4.3 項を参照のこと)
- アドレス演算子 (`&`) と間接参照 (`*`) (第 6.4.4 項を参照のこと)
- ビット否定 (1 の補数) (`~`) (第 6.4.5 項を参照のこと)
- キャスト演算子 (第 6.4.6 項を参照のこと)
- `sizeof` 演算子 (第 6.4.7 項を参照のこと)

6.4.1 単項プラス演算子と単項マイナス演算子

次の式を例にとります。

- 式

これはオペランドの符号の反転です。オペランドは算術型を持たなければならない、汎整数拡張が適用されます。`unsigned` 数の加算に関する逆元の場合には、`unsigned` 型の最大値に 1 を加えた値からその値を減算して計算されます。

単項プラス演算子は、次の式の値を返します。

+ 式

単項プラス演算子も単項マイナス演算子も、左辺値は生成しません。

6.4.2 否定

次の式を例にとります。

! 式

この結果は式の論理 (ブール) 否定になります。式の値が 0 である場合、その否定結果は 1 になります。式の値が 0 以外の場合、その否定結果は 0 になります。結果の型は `int` です。この式はスカラ型を持たなければなりません。

6.4.3 前置インクリメント演算子と前置デクリメント演算子

C 言語には、スカラ・オブジェクトの増分や減分を行うための 2 つの単項演算子があります。インクリメント演算子である `++` は、1 をオペランドに加算します。デクリメント演算子である `--` は、オペランドから 1 を減算します。`++` と `--` の両方とも、前置演算子 (変数の前、`++n`)、または後置演算子 (変数の後、`n++`) として使用できます。いずれの場合も `n` を増分します。式 `++n` は、その値が使用される前に `n` を増分します。これに対して、`n++` はその値が使用された後に `n` を増分します。

後置インクリメント演算子と後置デクリメント演算子については、第 6.3.4 項で説明しています。ここでは、前置形式について説明します。

次の式を例にとります。

`++` 可変左辺値

この式の評価後、結果は増分された右辺値であり、それに対応する左辺値ではありません。このため、この方法でインクリメント演算子とデクリメント演算子を使用する式は、左辺値が必要な代入式の左側にその式自体を記述することはできません。

オペランドを整数または浮動小数点数として宣言すると，1 (または 1.0) だけ増加または減少します。次に示すそれぞれの C 言語の文の結果は同じです。

```
i = i + 1;  
i++;  
++i;  
i += 1;
```

次に，インクリメント演算子の後置形式と前置形式の相違の例を示します。

```
int i, j;  
j = 5;  
i = ++j;           /* i = 6, j = 6 */  
i = j++;           /* i = 6, j = 7 */
```

オペランドがポインタである場合には，そのアドレスは整数値の 1 だけ増分されるのではなく，示されるオブジェクトのデータ型に決められているサイズ単位で増分されます。次にその例を示します。

```
char *cp;  
int *ip;  
++cp;              /* Incremented by sizeof(char) */  
++ip;              /* Incremented by sizeof(int)  */
```

次の式を例にとります。

-- 可変左辺値

前置演算子である -- は前置演算子である ++ と類似していますが，オペランドの値が減分される点が異なります。

インクリメント演算子とデクリメント演算子を使用する場合には，式の評価の順序に依存しないようにコーディングを行ってください。次の不明瞭な式を例にとります。

```
k = x[j] + ++j;
```

式と演算子

6.4 単項演算子

`x[j]` の `j` 値は `j` の増分前に評価されるのか、または増分後に評価されるのかが指定されていません。不明瞭さを避けるために、変数の増分を別の文で行います。次にその例を示します。

```
++j;  
k = x[j] + j;
```

6.4.4 アドレス演算子と間接参照

次の式を例にとります。

& 左辺値

この式の結果は左辺値のアドレスになります。左辺値は関数指名子か、または修飾されない配列識別子を含むオブジェクトを指定する左辺値です。register変数またはビット・フィールドは、左辺値にすることはできません。

次の式を例にとります。

```
*pointer
```

式がアドレスを解決すると、間接参照演算子 (`*`) を使用してそのアドレスに格納されている値にアクセスできます。

`*` のオペランドが関数名または関数ポインタの場合には、その結果は関数指名子になります。`*` のオペランドがオブジェクトへのポインタの場合には、その結果はそのオブジェクトを指定する左辺値になります。ポインタに無効な値 (例: 0) を代入した場合の `*` 演算の結果は定義されていません。

間接参照演算子である `*` は常に左辺値を生成します。アドレス演算子である `&` は左辺値を生成しません。

6.4.5 ビット否定

次の式を例にとります。

~ 式

この式の結果は、評価された式のビット否定 (1 の補数) になります。1 のビットはそれぞれ 0 のビットに変換され、逆に 0 のビットはそれぞれ 1 のビットに変換されます。式は整数型を持たなければなりません。コンパイラは通常の算術変換を行います (第 6.11.1 項を参照してください)。

6.4.6 キャスト演算子

キャスト演算子は、スカラ・オペランドを指定のスカラ・データ型または void に強制的に変換します。この演算子は、括弧で囲んだ型名を式の前に付けた構成です。次の式を例にとります。

(型名) 式

式の値は、指名されたデータ型に変換されます。これは、その式がその型の変数に代入された場合と同様に変換されます。この式の型と値自体は変更されませんが、キャスト演算の実行中だけその値がキャスト型に変換されます。型名の構文は、次のとおりです。

型名

型指定子 抽象宣言子

単純な場合は、型指定子は char または double などのデータ型のキーワードであり、抽象宣言子は空になります。次にその例を示します。

```
(int)x;
```

型指定子は、enum指定子または typedef名にすることもできます。型指定子は、抽象宣言子がポインタの場合にのみ、構造体または共用体にすることができます。つまり、型名は構造体や共用体へのポインタにすることはできますが、構造体と共用体はスカラ型ではないため、構造体や共用体にすることはできません。次にその例を示します。

式と演算子

6.4 単項演算子

```
(struct abc *)x /* allowed */
(struct abc)x   /* not allowed */
```

キャスト演算子の抽象宣言子は識別子のない宣言子です。抽象宣言子の構文は、次のとおりです。

抽象宣言子

```
空
抽象宣言子
* 抽象宣言子
抽象宣言子
抽象宣言子 [ 定数式 ]
```

次の形式では、抽象宣言子は空にできません。

(抽象宣言子)

抽象宣言子には、配列と関数を示す大括弧と括弧を含めることができます。ただし、キャスト演算は式を強制的に配列、関数、構造体、または共用体に変換することはできません。大括弧と括弧は、次の例に示すような演算に使用されます。この例では、P1識別子を `int` の配列へのポインタにキャストします。

```
(int (*)[10]) P1;
```

この種のキャスト演算は、P1 の内容を変更しません。この演算により、コンパイラは P1 の値をこの配列へのポインタとみなします。この方法でポインタをキャストすると、整数がポインタに加算されるときに起こるスケーリングを変更できません。

```
int *ip;
((char *)ip) + 1; /* Increments by 1 not by 4 */
```

キャスト演算子は、ポインタを含む次の変換に使用できます。

- ポインタを汎整数型に変換できます。ポインタは、`int`型または `long`型 (またはそれらの `unsigned` の同値) のオブジェクトと同じ記憶容量を占有します。したがって、ポインタはこれらの整数型のいずれかに変換でき、その値を変更せずに元に戻すことができます。スケーリングは起こらず、値の表現は変更されません。

ポインタから短整数型への変換は、unsigned long型から短整数型への変換に類似しています。つまり、ポインタの上位ビットが破棄されます。

短整数型からポインタへの変換は、短整数型から unsigned long型のオブジェクトへの変換に類似しています。つまり、ポインタの上位ビットは符号ビットのコピーで埋められます。エラー検査オプションを指定した *Compaq C* は、この型のキャスト演算に対して警告メッセージを出します。

- オブジェクトまたは不完全型へのポインタは、異なるオブジェクトまたは異なる不完全型へのポインタに変換できます。ポインタが示す型に対して不適切に境界調整されると、そのポインタが有効でなくなる場合があります。ただし、任意の境界調整のオブジェクトへのポインタは、同じ境界調整またはそれよりも厳密ではない境界調整のオブジェクトへのポインタに変換でき、それを元に戻せることが保証されています。この結果は元のポインタに等しくなります。文字型のオブジェクトは、最も厳密ではない境界調整を持ちます。
- ある型の関数に対するポインタを別の型の関数に対するポインタに変換でき、再度それを元に戻せます。この結果は、元のポインタに等しくなります。変換されたポインタを、呼び出された関数の型と互換性のない型の関数を呼び出す際に使用した場合の動作結果は定義されていません。

6.4.7 sizeof 演算子

次に、sizeof演算子の構文を示します。

sizeof 式

sizeof (型名)

型名は不完全型、関数型、またはビット・フィールドにすることはできません。sizeof演算子は、コンパイル時の整数定数値を生成します。式はその型を推定するためにのみ検査されます。つまり、式は完全には評価されません。たとえば、sizeof(i++) は sizeof(i) に等しくなります。

sizeof演算の結果は、オペランドのバイト単位のサイズになります。1 番目の場合、sizeof の結果は式の型で決定されるサイズになります。2 番目の場合、結果は指名された型のオブジェクトのサイズになります。sizeof の優先順位は他のほ

とんどの演算子よりも高いため、式が演算子を含む場合には、その式を括弧に入れなければなりません。

型名の構文はキャスト演算子の構文と同じです。次にその例を示します。

```
int x;  
x = sizeof(char *); /* assigns the size of a character pointer to x */
```

sizeof演算子の結果である `size_t` 型は符号なし整数型になります。 *Compaq C* では、`size_t` は `unsigned int` になります。

6.4.8 `__typeof__` 演算子

式の型を参照するもう 1 つの方法は、`__typeof__` 演算子です。この機能により、gcc コンパイラとの互換性が保たれます。

この演算子のキーワードの構文は `sizeof` に似ていますが、その構造は意味的に、`typedef` で定義された型名と同じように動作します。

```
__typeof__(式)  
__typeof__(型名)
```

`__typeof__` の引数には、式と型名の 2 通りの記述方法があります。

式を使用する例を次に示します。この例では、`x` は `int` の配列であるとし、記述される型は、`int` です。

```
__typeof__(x[0])(1)
```

引数として型名を使用する例を次に示します。記述される型は、`int` へのポインタ型です。

```
__typeof__(int *)
```

`__typeof__` 構造は、`typedef` 名が使用できるすべての場所で使用できます。たとえば、宣言内、キャスト内、`sizeof` または `__typeof__` 演算子の内部などで使用できます。


```
__typeof__ (*x) y;    // Declares y with the type of what x points to.  
__typeof__ (*x) y[4]; // Declares y as an array of such values.  
__typeof__ (__typeof__ (char *) [4]) y; // Declares y as an array of  
                                         // pointers to characters:
```

最後の例 (ネストした `__typeof__` 演算子) は、次のような従来の C 宣言と等価です。

```
char *y[4];
```

`__typeof__` を使用した宣言の意味と、そのように記述することがなぜ便利なのかを理解するために、マクロを使用して書き直してみます。

```
#define pointer(T) __typeof__(T *)  
#define array(T, N) __typeof__(T [N])
```

宣言は、次のように書き直せます。

```
array (pointer (char), 4) y;
```

このため、`array (pointer (char), 4)` は、`char` へのポインタが 4 つある配列の型になります。

6.4.9 `_Pragma` 演算子

`_Pragma` 演算子は、引数の文字列リテラルを非文字列化し、マクロ展開で効果的に `#pragma` 命令を生成します。この演算子を使って指定したときには、`_m` 接尾語が付いたトークンがこの形式の単一の文字列リテラル内に一緒に指定され、`_m` 接尾語が付いていたとしてもマクロ展開はされません。ただし、必要な場合は、文字列化演算子 (`#`) を使って文字列を形成することで、マクロ展開を実行させることができます (第 8.1.3 項を参照してください)。

`_Pragma` 演算子の構文は、次のとおりです。

```
_Pragma ( string-literal )
```

`_Pragma`演算子の式は、次のように処理されます。L 接頭語があればそれを削除し、先頭と末尾の二重引用符を削除し、エスケープ・シーケンス `\` を二重引用符で置き換え、エスケープ・シーケンス `\\` をバックスラッシュ 1 個に置き換えることで、文字列リテラルが非文字列化されます。

結果として得られた文字シーケンスは、翻訳フェーズ 3 で処理され、プラグマ命令の `pp-tokens` のように実行される前処理トークンが生成されます。単項演算子の式にある元の 4 つの前処理トークンは削除されます。

例: 次の形式のプラグマ命令があるとします。

```
#pragma listing on "..\listing.dir"
```

このプラグマ命令は、次のように表現することもできます。

```
_Pragma ( "listing on \"..\listing.dir\"" )
```

後者の形式は、そのまま記述した場合も、次のようなマクロ置換の結果として得られた場合も、同じ方法で処理されます。

```
#define LISTING(x) PRAGMA(listing on #x)
#define PRAGMA(x) _Pragma(#x)

LISTING ( ..\listing.dir )
```

6.5 2項演算子

次に、2項演算子の種類を示します。

- 乗除演算子 — 乗算 (`*`)、剰余 (`%`)、除算 (`/`) (第 6.5.1 項を参照のこと)
- 加減演算子 — 加算 (`+`) と減算 (`-`) (第 6.5.2 項を参照のこと)
- シフト演算子 — 左シフト (`<<`) と右シフト (`>>`) (第 6.5.3 項を参照のこと)
- 関係演算子 — 左不等号 (`<`)、等価左不等号 (`<=`)、右不等号 (`>`)、等価右不等号 (`>=`) (第 6.5.4 項を参照のこと)
- 等価演算子 — 等価 (`=`) と非等価 (`!=`) (第 6.5.5 項を参照のこと)
- ビット演算子 — AND (`&`)、OR (`|`)、XOR (`^`) (第 6.5.6 項を参照のこと)

- 論理演算子 — AND (&&) と OR (||) (第 6.5.7 項を参照のこと)

これ以降の項では、それぞれの 2 項演算子について説明します。

6.5.1 乗除演算子

乗除演算子は `*`、`/`、`%` です。オペランドは算術型を持たなければなりません。必要に応じて、オペランドは通常の算術変換規則により変換されます (第 6.11.1 項を参照してください)。

`*` 演算子は乗算を行います。

`/` 演算子は除算を行います。整数同士の除算の場合には、0 に近い値に切り捨てられます。いずれかのオペランドが負である場合には、その結果は 0 に近い値 (代数の商よりも絶対値の小さい最大整数) に切り捨てられます。

`%` 演算子は最初のオペランドを 2 番目のオペランドで割り、剰余を出します。両方のオペランドは汎整数でなければなりません。両方のオペランドが符号なしの数または正の数である場合、その結果は正となります。いずれかのオペランドが負である場合、結果の符号は左オペランドの符号と同じになります。

`b` が 0 以外の場合、次の文は真になります。

```
(a/b)*b + a%b == a;
```

エラー検査オプションを指定した場合、*Compaq C* コンパイラは定義されていない次の動作に対して警告を出します。

- 整数オーバーフロー
- 0 による除算
- 0 による剰余

6.5.2 加減演算子

加減演算子 `+` と `-` は、それぞれ加算と減算を行います。必要に応じて、通常の算術変換規則によりオペランドが変換されます (第 6.11.1 項を参照してください)。

2 つの `enum` 定数または変数が加算または減算されると、その結果は `int` 型になります。

整数がポインタ式に加算またはポインタ式から減算されると、その整数は示されるオブジェクトのサイズでスケーリングされます。この結果はポインタの型を持ちます。次にその例を示します。

```
int arr[10];
int *p = arr;
p = p + 1; /* Increments by sizeof(int) */
```

配列ポインタは、ポインタまたはアドレスから整数値を減算することにより減分できます。同様の変換が加算にも適用されます。また、ポインタ演算は配列の終わりを越えた次の 1 要素にも適用されます。たとえば、次のコーディングはポインタ演算が配列の要素とその配列を超えた次の 1 要素までに限定されているため有効です。

```
int i = 0;
int x[5] = {0,1,2,3,4};
int y[5];
int *ptr = x;
while (&y[i] != (ptr + 5)) { /* ptr + 5 marks one beyond the end of the array */
    y[i] = x[i];
    i++;
}
```

同じ配列の要素への 2 つのポインタ間で減算されると、その結果 (2 つのアドレス間の差を 1 つの要素の長さで割って計算した結果) は `ptrdiff_t` 型の値になります。これは *Compaq C* では `int` 型になり、アドレス指定された 2 つの要素間の要素の数を表します。2 つの要素が同じ配列にない場合の演算結果は定義されていません。

6.5.3 シフト演算子

シフト演算子である `<<` と `>>` は、右オペランドで指定したビット数だけ左オペランドを左 (`<<`) または右 (`>>`) にそれぞれシフトします。両方のオペランドは、整数でなければなりません。コンパイラは、各オペランドに対して汎整数拡張を行います (第 6.11.1.1 項を参照してください)。その結果の型は、拡張された左オペランドの型になります。次の式を例にとります。

E1 << E2

この式の結果は、*E2* ビットだけ左にシフトされた式 *E1* の値になります。シフトにより左端から押し出されたビットは破棄されます。空になった右のビットには 0 が埋め込まれます。左にシフトすることは、シフトする各ビットごとに左オペランドに 2 を掛けることになります。次の例では、*i* の値は 100 になります。

```
int n = 25;  
int m = 2;  
int i;  
  
i = n << m;
```

次の式を例にとります。

E1 >> E2

この式の結果は、*E2* ビットだけ右にシフトされた式 *E1* の値になります。シフトにより右端から押し出されたビットは破棄されます。*E1* が `unsigned` であるか、または *E1* が符号付きの型であるが負以外の値を持っている場合には、空になった左のビットには 0 が埋め込まれます。*E1* が符号付きの型で負の値を持っている場合には、空になったビットには 1 が埋め込まれます。

右オペランドが負の場合、またはその値が `int` のビット数よりも大きい場合のシフト演算結果は定義されていません。

左オペランドが負以外の場合、右にシフトすることはシフトする各ビットごとに左オペランドを 2 で割ることになります。次の例では、*i* の値は 12 になります。

式と演算子

6.5 2 項演算子

```
int n = 100;
int m = 3;
int i;

i = n >> m;
```

6.5.4 関係演算子

関係演算子は 2 つのオペランドを比較して、int 型の結果を生成します。関係が偽である場合には結果は 0 になり、真である場合には 1 になります。関係演算子には、左不等号演算子 (<)、右不等号演算子 (>)、等価左不等号演算子 (<=)、等価右不等号演算 (>=) があります。両方のオペランドとも算術型であるか、または互換型へのポインタでなければなりません。コンパイラは、比較を行う前に必要な算術変換を行います (第 6.11.1 項を参照してください)。

2 つのポインタを比較した結果は、各ポインタが示す 2 つのオブジェクトの相対位置によって異なります。下位アドレスのオブジェクトへのポインタは、上位アドレスのオブジェクトへのポインタよりも小さくなります。2 つのアドレスが同じ配列の要素を示す場合には、下位の添字を持つ要素のアドレスが上位の添字を持つ要素のアドレスよりも小さくなります。

関係演算子は左から右に結合されます。したがって、次に示す文は a を b に結合します。a が b より小さい場合、その結果は 1 (真) になります。a が b よりも大きいかまたは等しい場合、その結果は 0 (偽) になります。その後、その 0 または 1 が c と比較されて、式の結果が出ます。この文では、「b が a と c の間であるかどうか」は決定されません。

```
if ( a < b < c )
    statement;
```

b が a と c の間にあるかどうかを検査するには、次のようにコーディングします。

```
if ( a < b && b < c )
    statement;
```

6.5.5 等価演算子

等価演算子には等価 (==) と非等価 (!=) があり, int 型の結果を生成します。したがって, 次の文の結果は両方のオペランドが同じ値を持つ場合は 1, 持たない場合は 0 になります。

```
a == b
```

2 つのオペランドは, 次を示す型の組み合わせのいずれかでなければなりません。

- 両方のオペランドが算術型を持つ。
- 両方のオペランドが互換型の修飾型または未修飾型へのポインタである。
- 一方のオペランドがオブジェクト型または不完全型へのポインタであり, もう一方が void の修飾型または未修飾型へのポインタである。
- 1 つのオペランドがポインタであり, もう一方が空ポインタ定数である。

オペランドは必要に応じて, 通常の算術変換規則により変換されます (第 6.11.1 項を参照してください)。

2 つのポインタまたはアドレスが同じ記憶位置を識別する場合, その 2 つは等しいものです。

注意

代入 (=) と等価 (==) にはそれぞれ異なる記号が使用されますが, C 言語では両方の演算子ともすべての文脈で使用することができます。したがって, この 2 つの演算子を混同しないように注意してください。次にその例を示します。

```
if ( x = 1 )  
    文 1;  
else  
    文 2;
```

この例では, 文 1 が必ず実行されます。これは, 代入 x=1 の結果が x の値と同等であり, x には常に 1 (つまり真) が代入されるからです。

6.5.6 ビット演算子

ビット演算子には汎整数オペランドが必要です。通常の算術変換が行われます (第 6.11.1 項を参照してください)。式の結果は、2 つのオペランドに関するビット AND (&), 包含 OR (|), または排他的 OR (^) になります。オペランドの評価の順序は保証されません。

オペランドはビット単位で評価されます。1 つのビット値が 0 でもう一方が 1 である場合、または両方のビット値が 0 である場合、& 演算子の結果は 0 になります。両方のビット値が 1 である場合、その結果は 1 になります。

両方のビット値が 0 である場合、| 演算子の結果は 0 になります。いずれかのビット値が 1 である場合、または両方のビット値が 1 である場合、その結果は 1 になります。

両方のビット値が 0 である場合、または両方のビット値が 1 である場合、^ 演算子の結果は 0 になります。いずれかのビット値が 1 で、もう一方が 0 である場合、その結果は 1 になります。

6.5.7 論理演算子

論理演算子には、AND (&&) と OR (||) があります。この演算子は左から右に評価されます。式の結果 (int 型) は、0 (偽) または 1 (真) のいずれかです。2 つのオペランドは同じ型を持つ必要はありませんが、両方の型ともスカラ型でなければなりません。コンパイラの評価が左オペランドのみを検査することにより行われる場合には、右オペランドは評価されません。次の式を例にとります。

E1 && E2

両方のオペランドが 0 以外の場合、この式の結果は 1 になります。またはオペランドの一方が 0 である場合、式の結果は 0 になります。式 *E1* が 0 である場合、式 *E2* は評価されません。これは、*E2* の値とは無関係にその結果が同じであるからです。

同様にいずれかのオペランドが 0 以外の場合，次の式は 1 になり，それ以外の場合は 0 になります。式 $E1$ が 0 以外の場合には， $E2$ は評価されません。これは， $E2$ の値とは無関係に結果が同じになるからです。

$E1 \ || \ E2$

6.6 条件演算子

条件演算子 ($?:$) は，3 つのオペランドをとります。この演算子は最初のオペランドの結果を検査し，次に最初の結果に基づいて残りの 2 つのオペランドのいずれか 1 つを評価します。次にその例を示します。

$E1 \ ? \ E2 \ : \ E3$

式 $E1$ が 0 以外の (真) 場合は $E2$ が評価され，それが条件式の値になります。 $E1$ が 0 の (偽) 場合は $E3$ が評価され，それが条件式の値になります。条件式は右から左に結合されます。次の例では，条件演算子は x と y の最小値を得るために使用されます。

$a = (x < y) ? x : y; \quad /* \ a = \min(x, y) \ */$

最初の式 ($E1$) の後に評価順序点があります。次に示す例では結果が予測できるため，予定外の副作用が起こる可能性はありません。

$i++ > j ? y[i] : x[i];$

条件演算子は左辺値を生成しません。したがって， $a ? x : y = 10$ のような文は有効ではありません。

条件演算子には，次の制限があります。

- 最初のオペランドは，スカラ型を持たなければならない。

- 2 番目のオペランドと 3 番目のオペランドには、次の制限のいずれか 1 つが適用されなければならない。
 - 両方のオペランドが算術型を持つ。2 番目のオペランドと 3 番目のオペランドを共通の型にするために、通常の算術変換が行われる。結果はその型になる。
 - 両方のオペランドが互換性のある構造体型または共用体型を持つ。
 - 両方のオペランドが void 型を持つ。
 - 両方のオペランドが、互換型の修飾型または未修飾型へのポインタである。結果は合成型になる。
 - 一方のオペランドがポインタであり、もう一方が空ポインタ定数である。結果の型は、空ポインタ定数以外のポインタの型になる。
 - 一方のオペランドが、オブジェクト型または不完全型へのポインタであり、もう一方が void の修飾型または未修飾型へのポインタである。結果の型は void へのポインタ型になる。

6.7 代入演算子

代入演算子にはいくつか種類があります。代入を行った結果の値は、その変数に入ります。代入は、より大きい式の部分式として使用できます。代入演算子は左辺値を生成しません。

代入式は 2 つのオペランド、すなわち左に可変左辺値および右に式を持ちます。単純代入は、次に示すように 2 つのオペランドの間に等号 (=) を入れます。

```
E1 = E2;
```

式 E2 の値は E1 に代入されます。型は E1 の型になり、演算終了後の結果は E1 の値になります。

複合代入は、2 つのオペランドの間に等号 (=) の前か後に別の 2 項演算子を組み合わせたものを入れます。次にその例を示します。

```
E1 += E2;
```

この式は次に示す単純代入と同等です。ただし、複合代入では E1 は 1 回評価されるのに対して、単純代入では E1 が 2 回評価されます。

```
E1 = E1 + E2;
```

次に示す 2 つの式では、代入は等しくなります。

```
a *= b + 1;  
a = a * (b + 1);
```

この例では、次に示す式は number[1] の定数に 100 を加算します。

```
number[1] += 100;
```

この式の結果は加算後の値になり、number[1] と同じ型を持ちます。

両方の代入オペランドが算術型である場合、右オペランドは代入前に左の型に変換されます (第 6.11.1 項を参照してください)。

代入演算子 (=) は、構造体と共用体に値を代入するために使用できます。*Compaq C* の VAX C 互換性モードでは、2 つの構造体がバイト単位で同じサイズに定義されている限り、一方の構造体をもう一方の構造体に代入できます。ANSI 規格モードでは、2 つの構造体は同じ型を持たなければなりません。すべての複合代入演算子では、すべての右オペランドと左オペランドは、ポインタであるか、または算術値に評価されなければなりません。演算子が -= または += である場合には、左オペランドをポインタにすることができ、(汎整数である) 右オペランドは、2 項プラス演算子 (+) と 2 項マイナス演算子 (-) の右オペランドと同じ方法で変換されます。

複合代入演算子を構成する文字は、前後を逆に記述してはなりません。次にその例を示します。

```
E1 += E2;
```

この例は古い形式で現在はサポートされていませんが、エラーが検出されずにコンパイラを通過します。この場合、代入演算子の後に単項プラス演算子が続くものと解釈されます。

6.8 コンマ演算子

コンマ演算子で区切った複数の式は、左から右に評価されます。結果は、(他の式の副作用が起こることがあっても) 一番右側の式の型と値になります。結果は左辺値ではありません。次の例では 1 が R に代入され、2 が T に代入されます。

```
R = T = 1,    T += 2,    T -= 1;
```

それぞれの式の副作用は、次の式が評価される前に完了します

コンマが別の意味で実引数や初期化並びに現れる場合には、コンマ式は括弧で囲まなければならないかもしれません。次にその例を示します。

```
f(a, (t=3,t+2), c)
```

この例では、a, 5, および c 実引数を持つ f 関数を呼び出します。さらに、t 変数に 3 を代入します。

6.9 定数式

定数式とは定数のみを含む式のことです。定数式は実行時ではなくコンパイル時に評価でき、定数を記述できる任意の場所で使用できます。次の例では limit+1 が定数式であり、コンパイル時に評価されます。

```
#define limit 500  
char x[limit+1]
```

定数式には、代入演算子、インクリメント演算子、デクリメント演算子、関数呼出し演算子、またはコンマ演算子を含めることはできません。ただし、それらが sizeof 演算子のオペランド内にある場合は例外です。各定数式は、その型で表現可能な値の範囲内にある定数を評価しなければなりません。

次に、*Compaq C*において定数を評価する式が必要な文脈を示します。

- ビット・フィールドのサイズ
- 列挙定数の値

- 配列 (および配列宣言内の 2 次元以上) のサイズ
- case ラベルの値
- 条件付き取込みの前処理命令に使用される汎整数定数式
- 静的記憶域存続期間を持つオブジェクトの初期化子並び

6.9.1 汎整数定数式

汎整数定数式は汎整数型を持ち、オペランドは整数定数、列挙定数、文字定数、オペランドに可変長配列型や括弧で囲んだ可変長配列型名を持たない sizeof 式、またはキャストの直接のオペランドである浮動小数点定数でなければなりません。汎整数定数式のキャスト演算子は、sizeof 演算子のオペランドの一部である場合以外は、算術型のみを汎整数型に変換します。

Compaq C では、初期化子の定数式の許容範囲を拡大することができます。この定数式では、次のいずれかを評価できます。

- 算術定数式
- 空ポインタ定数
- アドレス定数
- オブジェクト型のアドレス定数に汎整数定数式をプラスまたはマイナスしたもの

6.9.2 算術定数式

算術定数式は算術型の式であり、整数定数、浮動小数点定数、列挙定数、文字定数、または、オペランドに可変長配列型や括弧で囲んだ可変長配列型名を持たない sizeof 式のオペランドのみを含みます。算術定数式のキャスト演算子は sizeof 演算子のオペランドの一部である場合以外は、算術型のみを算術型に変換します。

6.9.3 アドレス定数

アドレス定数は、静的記憶域存続期間 (第 2.10 節を参照のこと) のオブジェクトを指定する左辺値へのポインタ、または関数指名子へのポインタです。アドレス定数は単項&演算子を使用して明示的に作成するか、または配列型か関数型の式を使用して暗黙に作成しなければなりません。配列添字 [] およびメンバ・アクセス演算子の . と -> , アドレス演算子 & と間接指定単項演算子 * , およびポインタ・キャストを使用してアドレス定数を作成できます。しかし、これらの演算子を使用してオブジェクトの値にアクセスすることはできません。

6.10 複合リテラル式

複合リテラルは構成子式(constructor expression) と呼ばれ、配列、構造体、共用体型などのオブジェクトの値を構成する式です。

C89 標準では、structの値を関数に渡す場合、通常は、その型の名前付きオブジェクトを宣言し、そのメンバを初期化し、そのオブジェクトを関数に渡します。C99 標準では、この処理を 1 つの複合リテラル式で実行できるようになりました。(複合リテラル式は、*Compaq C* コンパイラのコモン C、VAX C、Strict ANSI89 モードではサポートされていないので注意してください。)

複合リテラルは名前なしオブジェクトであり、括弧で囲んだ型名 (キャスト演算子¹と同じ構文) の後に中括弧で囲んだ初期化子の並びを続けるという構文で指定されます。この名前なしオブジェクトの値は、初期化子リストで決定されます。初期化子の並びには、指名子の構文を使用できます。

たとえば、配列要素 9 の値が 5 で、それ以外の値はすべてゼロである 1000 個の int の配列を作成するには、次のように記述します。

```
(int [1000]){[9] = 5}.
```

¹ ただし、キャストがスカラ型または void のみへの変換を指定するキャスト式とは異なり、キャスト式の結果は左辺値ではありません。

複合リテラルのオブジェクトは左辺値です。複合リテラルが指名するオブジェクトは、すべての関数定義の外にある場合、静的記憶域存続期間を持ちます。すべての関数定義の外でない場合は、囲んでいる最も近いブロックに対応する自動記憶域存続期間を持ちます。

使用上の注意

- 型名には、オブジェクト型、またはサイズ不明の配列を指定しなければならない。
- 初期化子は、複合リテラルで指定される名前なしオブジェクト内に含まれないオブジェクトの値を指定することはできない。
- 複合リテラルが関数本体の外部にある場合、初期化子並びは定数式で構成されていなければならない。
- サイズ不明の配列を型名が指定する場合、サイズは初期化子の並びによって決まり (第 4.7.1 項を参照)、複合リテラルの型は完成後の配列の型になる。それ以外の場合 (型名がオブジェクト型を指定する場合)、複合リテラルの型は、型名によって指定された型になる。いずれの場合も、結果は左辺値になる。
- 4.2, 4.7.1, 4.8.4, 4.8.5, 4.9 に記載した、初期化子の並びに対する意味上の規則と制限は、複合リテラルにもすべて適用できる。
- 文字列リテラルと、const-qualified 型の複合リテラルは、異なるオブジェクトである必要はない。これにより、処理系では、同一または重複する表現の文字列リテラルと定数複合リテラルの間で、記憶域を共有することができる。

次の例では、複合リテラルの使用方法を示します。

例

1. `int *p = (int []){2, 4};`

この例では、配列の最初の要素を指すようにpを初期化します。この配列にはintが2つあり、1番目の値は2で、2番目の値が4です。この複合リテラルの中の式は、定数でなければなりません。この名前なしオブジェクトは、静的記憶域存続期間を持ちます。

```
2. void f(void)
   {
       int *p;
       /*...*/
       p = (int [2]){*p};
       /*...*/
   }
```

この例では、pには配列の最初の要素のアドレスが代入されます。この配列にはintが2つあり、1番目の値はpが直前に指していた値で、2番目の値はゼロです。この複合リテラルの式は、定数でなくても構いません。この名前なしオブジェクトは、自動記憶域存続期間を持ちます。

```
3. drawline((struct point){.x=1, .y=1},
            (struct point){.x=3, .y=4});
```

または、drawlineがpoint構造体へのポインタを必要とする場合は、次のようになります。

```
drawline(&(struct point){.x=1, .y=1},
        &(struct point){.x=3, .y=4});
```

ディジグネーションを持つ初期化子は、複合リテラルと組み合わせることができます。複合リテラルを使用して作成した構造体オブジェクトは、メンバの順序とは無関係に関数に渡すことができます。

```
4. (const float []){1e0, 1e1, 1e2, 1e3, 1e4, 1e5, 1e6}
```

読取り専用の複合リテラルは、この例のような構文で指定できます。

5.

```
"/tmp/testfile"  
(char []){"/tmp/testfile"}  
(const char []){"/tmp/testfile"}
```

この例の3つの式は、意味が異なります。

最初の式は、記憶域存続期間が常に静的であり、型は「charの配列」ですが、変更可能であるとは限りません。

後の2つは、関数本体の内部にある場合は記憶域存続期間が自動であり、前者は変更が可能です。

6.

```
(const char []){"abc"} == "abc"
```

文字列リテラルと同じように、constで修飾した複合リテラルは、読取り専用メモリに配置でき、共用も可能です。このリテラルの記憶域が共用されている場合、この例の結果は1になります。

7.

```
struct int_list { int car; struct int_list *cdr; };  
struct int_list endless_zeros = {0, &endless_zeros};  
eval(endless_zeros);
```

複合リテラルには名前がないため、単一の複合リテラルでは、循環してリンクされるオブジェクトを指定することはできません。この例で言えば、名前付きオブジェクトendless_zerosの代わりに関数の引数として使用できる、自己参照型の複合リテラルを記述する方法はありません。

```
8. struct s { int i; };  
int f (void)  
{  
    struct s *p = 0, *q;  
    int j = 0;  
    while (j < 2)  
        q = p, p = &((struct s){ j++ });  
    return p == q && q->i == 1;  
}
```

この例に示すように、複合リテラルはそれぞれ、指定されたスコープ中で*オブジェクトを1つだけ作成します。

f()関数は必ず値1を返します。

6.11 データ型変換

C言語では、次に示す4つの場合にデータ型変換を行います。

1. 異なる型の複数のオペランドが1つの式に現れる場合
2. char型, short型, および float型の実引数が旧形式の宣言を使用して関数に渡される場合
3. 関数プロトタイプで宣言される仮引数に厳密に一致していない実引数が関数に渡される場合
4. オペランドのデータ型が, キャスト演算子によって強制的に変換される場合 (キャスト演算子については, 第6.4.6項を参照のこと)

これ以降の項では、オペランドと関数の実引数の変換方法について説明します。

6.11.1 通常の算術変換

次に示す規則は通常の算術変換と呼ばれ、算術式のオペランドのすべての変換に適用されます。この算術変換ではオペランドを共通の型に変換し、この型が結果の型になります。規則は次の順序で適用されます。

1. いずれかのオペランドが算術型のオペランド以外の場合、変換は行われない。
2. いずれかのオペランドが `long double` 型を持つ場合、もう一方のオペランドは `long double` に変換される。
3. 前記以外でいずれかのオペランドが `double` 型を持つ場合、もう一方のオペランドは `double` に変換される。
4. 前記以外でいずれかのオペランドが `float` 型を持つ場合、もう一方のオペランドは `float` に変換される。
5. 複素数の場合 (*Alpha only*) , 上記の変換は、変換対象のオペランドに対応する実数型で行われる。

たとえば、`double _Complex` と `float` を足すと、`float` のオペランドは `double` に変換されるだけで、結果は `double _Complex` となる。

6. 前記以外の場合には汎整数拡張が両方のオペランドで実行され、次の規則が適用される。
 - a. 両方のオペランドの型が同じ場合、変換はそれ以上必要ない。
 - b. 型が同じでない場合、両方のオペランドが符号付き整数型であるか、両方のオペランドが符号なし整数型であれば、整数変換ランクの低いオペランドが、ランクの高いオペランドの型に変換される。
 - c. 前記以外の場合でいずれかのオペランドの型が `unsigned long long int` のとき、もう一方のオペランドは `unsigned long long int` に変換される。
 - d. いずれかのオペランドの型が `unsigned long int` の場合、もう一方のオペランドは `unsigned long int` に変換される。
 - e. 前記以外の場合で 1 つのオペランドが `long int` 型を持ち、もう一方のオペランドが `unsigned int` 型を持つとき、`long int` が `unsigned int` の値すべてを表すことができる場合、`unsigned int` 型のオペランドは `long int` に変換される。

換される。long int が unsigned int の値をすべて表すことができない場合、両方のオペランドは unsigned long int に変換される。

- f. 前記以外でいずれかのオペランドが long int型を持つ場合、もう一方のオペランドは long int に変換される。
- g. 前記以外でいずれかのオペランドが unsigned int型を持つ場合、もう一方のオペランドは unsigned int に変換される。
- h. 前記以外の場合、両方のオペランドが int型を持つ。

これ以降の項では、通常の算術変換規則について説明します。

6.11.1.1 文字と整数

符号付きまたは符号なしの charビット・フィールド、short intビット・フィールド、または intビット・フィールド、あるいは列挙型を持つオブジェクトは、int または unsigned int が使用可能な式に使用できます。int が元の型の値をすべて表すことができる場合には、その値は int に変換されます。それ以外の場合は unsigned int に変換されます。このような変換規則を汎整数拡張と呼びます。

この汎整数拡張の実現を値保存と呼びます。これに対して、unsigned char と unsigned short型を unsigned int に拡張することを符号なし保存と呼びます。*Compaq C* は、コモン C モードが指定されない限り ANSI C 規格で規定されている値保存の拡張を使用します。

エラー検査オプションを指定した *Compaq C* では、符号なし保存規則に依存する算術変換の検索を容易にするために unsigned char と unsigned short から int への汎整数拡張にフラグを立てます。これは、値保存の汎整数拡張により影響を受ける可能性があります。

その他のすべての算術型は、汎整数拡張により変更されることはありません。

Compaq C では、char型の変数は符号付き整数として処理されるバイトです。長い整数が短い整数または char に変換されると切り捨てが左端で行われ、超過したビットは破棄されます。次にその例を示します。

```
int i;  
char c;  
  
i = 0xFFFFFFFF41;  
c = i;
```

このコーディングでは、16 進の 41 ('A') を c に代入します。コンパイラは、符号拡張によって短い符号付き整数を長い整数に変換します。

6.11.1.2 符号付き整数と符号なし整数

変換はさまざまな種類の整数の間でも行われます。

汎整数型を持つ値を別の汎整数型に変換した場合 (例: int を long int に変換した場合)、値を新しい型で表すことができる場合にはその値は変更されません。

符号付き整数をそれ以上のサイズを持つ符号なし整数に変換したとき、その符号付き整数値が負以外の場合には値は変更されません。符号付き整数値が負である場合には次の変換が行われます。

- 符号なし整数型の方が大きい場合にはまず、その符号付き整数は符号なし整数に対応する符号付き整数に拡張されます。次に、その値に符号なし整数型で表せる最大数よりも 1 だけ大きい値を加算することにより、符号なし整数型に変換します。
- 符号なし整数型が符号付き整数型に等しいか、小さい場合にはその値に符号なし整数型で表せる最大数よりも 1 だけ大きい値を加算することにより、その値を符号なし整数型に変換します。

整数値を小さいサイズの符号なし整数に縮小する場合には、その結果はその新しい汎整数型で表せる最大の符号なしの値よりも大きい値で割った値の負以外の剰余になります。

整数値を小さいサイズの符号付き整数に縮小する場合、または符号なし整数をそれに対応する符号付き整数に変換する場合には、その値は新しい型で表すのに十分なサイズであれば変更されません。それ以外の場合には、結果は切り捨てられます。つまり、超過した上位ビットが破棄され、精度は損なわれます。

同じサイズの汎整数型間での変換の場合には、符号付きでも符号なしでもマシン・レベルの表現は変更されません。

6.11.1.3 浮動小数点と汎整数

浮動小数点型オペランドが整数に変換されると、小数部は切り捨てられます。

コンパイル時に浮動小数点型の値を整数型または別の浮動小数点型に変換したとき、その結果を表すことができない場合には、コンパイラは次に示す場合に対して警告を出します。

- unsigned int への変換で、結果を unsigned int 型で表すことができない場合
- unsigned int 以外の型への変換で、結果を int 型で表すことができない場合

汎整数型の値を浮動小数点型に変換したとき、その値が表現できる値の範囲内にはあるが厳密ではない場合には、変換の結果は次に大きい値または小さい値になります。いずれの値になるかは、各ハードウェアの変換結果により異なります。使用する各プラットフォームの変換結果については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

6.11.1.4 浮動小数点型

式の中に float 型のオペランドが含まれる場合には、そのオペランドは単精度オブジェクトとみなされます。ただし、その式に double 型または long double 型のオブジェクトも含まれる場合には、通常の算術変換が適用されます。

float が double か long double に拡張される場合、または double が long double に拡張される場合には、その値は変更されません。

double を float に縮小する場合、または long double を double か float に縮小する場合、変換する値が表現できる値の範囲外にあるとその動作結果は定義されていません。

変換する値が表現できる値の範囲内にあるが厳密ではない場合には、その結果は次に大きいか、または小さい表現可能な float 値に丸められます。

6.11.2 ポインタ変換

2つの型 (例: `int` と `long`) は同じ表現を持つことはできますが、それぞれ異なる型です。つまり、`int` へのポインタはキャストを使用せずに `long` へのポインタに代入することはできません。また、ある型の関数へのポインタはキャストを使用せずに異なる型の関数へのポインタに代入することもできません。さらに、異なる仮引数型情報 (仮引数型情報を持たない旧形式を含む) を持つ関数へのポインタの型は、異なる型です。このような場合にキャストを使用しなければ、コンパイラはエラーを出します。プロセッサの中には境界調整の制限を持つものがあるため、境界調整されていないポインタでアクセスすると、アクセス時間がかかったり、マシン例外が起こることがあります。

`void` へのポインタと、任意の不完全型またはオブジェクト型へのポインタは相互に変換できます。不完全型またはオブジェクト型へのポインタが `void` へのポインタに変換され、元に戻されると、その結果は元のポインタに等しくなります。

`void *` 型へキャストされた 0 に等しい汎整数定数式または式を、空ポインタ定数と呼びます。空ポインタ定数があるポインタに代入されるか、またはポインタに等しいかどうか比較される場合、その定数はその型のポインタに変換されます。このようなポインタを空ポインタと呼び、任意のオブジェクトまたは関数へのポインタと比較しても等しくなりません。

配列指名子は配列型へのポインタに自動的に変換され、そのポインタは配列の最初の要素を示します。

6.11.3 関数の実引数の変換

関数プロトタイプ宣言が存在しなければ、関数の実引数のデータ型は仮引数の型に一致するものとみなされます。関数プロトタイプが存在すると、関数呼出しの実引数がすべて関数プロトタイプ宣言に宣言されているすべての仮引数との代入互換性を持つかどうか比較されます。実引数の型が仮引数の型とは一致しないけれども代入互換性を持つ場合、C 言語はその実引数をその仮引数の型に変換します (第 6.11.1 項を参照してください)。関数呼出しの実引数が関数プロトタイプ宣言に宣言されている仮引数と代入互換性を持たない場合には、エラー・メッセージが出されます。

関数プロトタイプが存在しなければ、float型のすべての実引数が double に、char または short型のすべての実引数が int型に、unsigned char と unsigned short型のすべての実引数が unsigned int に変換されます。配列名または関数名は、指名された配列または関数のアドレスにそれぞれ変換されます。コンパイラはその他の変換を自動的には行いません。このような変換の後に不互換が生じた場合はプログラム・エラーです。

関数指名子は関数型を持つ式です。この指名子が sizeof演算子のオペランドまたは単項&演算子である場合以外は、「型を返す関数」型を持つ関数指名子は「型を返す関数へのポインタ」型を持つ式に変換されます。

この章では、C プログラミング言語における次の種類の文について説明します。この章で示す文以外の文は、関数本体に現れる順序で実行されます。

- ラベル付き文 (第 7.1 節)
- 複文 (第 7.2 節)
- 式文 (第 7.3 節)
- 空文 (第 7.4 節)
- 選択文 (第 7.5 節)
- 繰返し文 (第 7.6 節)
- 飛越し文 (第 7.7 節)

7.1 ラベル付き文

ラベルは、goto文または switch文の行き先として、プログラム内のある位置にフラグを立てる際に使用する識別子です。その構文は以下のとおりです。

識別子 : 文

case 定数式 : 文
default : 文

ラベルのスコープはそのラベルを含んでいる関数本体です。変数には、関数内のラベルと同じ名前をつけることができます。これは、ラベルと変数の名前空間が異なるからです。第 2.15 節を参照してください。

C 言語のラベル付き文には、次の 3 種類があります。

文

7.1 ラベル付き文

- ラベルが先行する文
- case文
- default文

case文と default文は，switch文の中にのみ指定します。第 7.5.2 項を参照してください。

7.2 複文

複文，すなわちブロックは一連の文を単一の文として扱うことができます。複文は左中括弧で開始し，宣言と文を任意の順で含み，右中括弧で終了します。次にその例を示します。

```
{  
    int a;  
    a = 1;  
    int b;  
    b = 2;  
}
```

注意

コモン C，VAX C，Strict ANSI89 モードでは，複文の中で宣言と文を任意の順序にすることはできません。これらのモードでは，宣言を最初に指定し，その後に文を続けなければなりません。

ブロック宣言は各ブロック内でのみ有効であり，そのブロック内では外側のスコープで宣言されている同じ名前の他の宣言に優先します。

制御フローがブロックに入るか，または goto文がそのブロックの始めのラベルに制御を渡すと，そのブロックは正常に開始されます。ブロックが正常に開始されるたびに，auto または register変数に対して記憶域が割り当てられます。これに対して goto文が制御をブロック内のラベルに渡した場合，またはブロックが switch文の本体である場合にはこの記憶域の割り当ては行われません。記憶域クラスについての詳細は，第 2.10 節を参照してください。

関数定義は複文を含んでいます。関数定義において、仮引数宣言に続く複文を関数本体と呼びます。

7.3 式文

式の後にセミコロンを付けることによって、有効な式を文として使用することができます。次にその例を示します。

```
i++;
```

この文は、`i` 変数の値を増分します。`i++` は C 言語の複雑な文にも現れる C 言語の有効な式です。C 言語の式についての詳細は、第 6 章を参照してください。

7.4 空文

空文は、C 言語の文法により文を必要とする場合に空演算を行う際に使用されますが、プログラム中で特に作業を行う必要はありません。空文はセミコロンで構成されます。

```
;
```

空文は `if`、`while`、`do`、および `for` 文を使用する際に役に立ちます。この文は主に、すべてのループ処理がループの判定部分で実行される場合にループ演算において使用されます。たとえば、次の文では配列中の値が 0 の最初の要素を見つけます。

```
for (i=0; array[i] != 0; i++)
    ;
```

この例の `for` 文はその副作用のためにだけ実行され、ループ本体は空文です。繰返し文についての詳細は、第 7.6 節を参照してください。

空文は、複文を終了させる中括弧の直前にラベルが必要な場合にも役立ちます。ラベルは、右中括弧の直前に付けることはできません。必ず文に付加する必要があります。次にその例を示します。

文

7.4 空文

```
if (expression1)
{
    ...
    goto label_1; /* Terminates this part of the if statement */
    ...
label_1: ;
}
else ...
```

7.5 選択文

選択文は制御式の値に応じて一連の文の中から文を選択します。選択文には if文と switch文があります。これ以降の項では、これらの文について説明します。

7.5.1 if 文

if文の構文は次のとおりです。

if (式)

文

else_{opt}

文_{opt}

制御式の値が真 (0 以外) の場合には、制御式に続く文が実行されます。if文にオプションの else句を付けて記述すると、制御式の値が偽 (0) の場合にこの else句が実行されます。

次にその例を示します。

```
if (i < 1)
    funct(i);
else
{
    i = x++;
    funct(i);
}
```

この例では、*i* の値が 1 未満の場合には `funct(i)` 文が実行され、`else` キーワードに続く複文は実行されません。*i* の値が 1 未満ではない場合には、`else` キーワードに続く複文だけが実行されます。

選択文中の制御式は通常は論理式ですが、スカラ型であればどんな式でも構いません。

`if` 文がネストされている場合、`else` 句は同じブロック内にある `else` 句を持たない直前の `if` 文と対になります。次にその例を示します。

```
if (i < 1)
{
    if (j < 1)
        funct(j);
    if (k < 1)          /* This if statement is associated with */
        funct(k);
    else                /* this else clause.                  */
        funct(j + k);
}
```

7.5.2 switch 文

`switch` 文は、制御式の値に基づいて 1 つまたは複数の一連の文を実行します。

`switch` 文の構文は次のとおりです。

`switch`

文

文

7.5 選択文

通常の算術型変換は制御式で行われますが、その結果は汎整数型でなければなりません。データ型変換についての詳細は第 6.11 節を参照してください。switch文は一般に複文であり、制御式が caseラベルと一致した場合に 1 つまたは複数の case文が実行されます。

caseラベルおよび式の構文は次のとおりです。

case 定数式 : 文

定数式は汎整数型でなければなりません。2 つの caseラベルに同じ値を指定することはできません。switch文中の caseラベルの数に制限はありません。

複文中で次のラベルを持つことができるのは、1 つの文だけです。

default :

case および defaultラベルの順序は決まっていますが、通常は case文の後に default文を続けます。break文を指定しない限り、選択した文から順番に処理が実行されます。

switch文を実行すると、次の順序で処理が行われます。

1. switch制御式が評価され (さらに汎整数拡張が適用され)、caseラベル中の定数式と比較されます。
2. 制御式の値が caseラベルと一致した場合には、制御はこのラベルに続く文へ渡されます。break文が見つかり、switch文は終了します。それ以外の場合には、実行は次の case文または default文へ続き、break文または switch文の終わりが見つかりと終了します (例 7-1 を参照してください)。

switch文は、return文または goto文でも終了することができます。switch文がループの内側にある場合には、continue文がループを終了すると switch文は終了します。これらの文についての詳細は、第 7.7 節を参照してください。

3. 制御式の値がどの caseラベルとも一致せず、defaultラベルがある場合に制御はこのラベルに続く文へ渡されます。break文で default文が終了せず、caseラベルが続く場合はその case文が実行されます。
4. 制御式の値がどの caseラベルとも一致せず、defaultラベルがない場合には switch文の実行は終了します。

例 7-1 では、switch文を使用して、ターミナルから入力した空白、タブ、および改行文字の数をカウントします。

例 7-1 switch 文を使用した空白、タブ、および改行のカウント

```

/* This program counts blanks, tabs, and new lines in text *
 * entered from the keyboard.                               */

#include <stdio.h>
main()
{
    int number_tabs = 0, number_lines = 0, number_blanks = 0;
    int ch;
    while((ch = getchar()) != EOF)
        switch (ch)
        {
1       case '\t': ++number_tabs;
2       break;
          case '\n': ++number_lines;
          break;
          case ' ': ++number_blanks;
          break;
          default;;
        }
    printf("Blanks\tTabs\tNewlines\n");
    printf("%6d\t%6d\t%6d\n", number_blanks,
        number_tabs,number_lines);
}

```

次は、例 7-1 の説明です。

- 1 一連の case文を使用して、入力された文字に応じて別々のカウンタを増分します。
- 2 break文によって、制御は whileループに戻ります。ch の値がどの case定数式とも一致しない場合には、制御は whileループへ渡されます。

break文がない場合には、次の case文が続けて実行されます。

文 7.5 選択文

変数宣言が switch文の中の複文に現れた場合には、auto または register宣言への初期値は無効になります。ただし、case文に続く文の中の初期化は有効です。次にその例を示します。

```
switch (ch)
{
    int nx = 1;          /* Initialization ignored      */
    printf("%d", n);     /* This first printf is not executed */
    case 'a' :
        { int n = 5;     /* Proper initialization occurs */
          printf("%d", n);
          break; }
    case 'b' :
        { break; }
    default:
        { break; }
}
```

この例で `ch == 'a'` の場合はプログラムは 5 を出力します。変数が他の文字と等しい場合にはプログラムは何も出力しません。これは、caseラベルの外側の初期化および文は無効だからです。

7.6 繰り返し文

繰り返し文，すなわちループは制御式が偽 (0) になるまでループ本体の文を繰り返し実行します。制御式はスカラ型でなければなりません。

while文は、ループ本体を実行する前に制御式を評価します (第 7.6.1 項を参照してください)。

do文は、ループ本体を実行した後に制御式を評価します。したがって、ループ本体は必ず 1 回は実行されます (第 7.6.2 項を参照してください)。

for文は、3 つの式の中の 2 番目の式の評価に基づいてループ本体を実行します (第 7.6.3 項を参照してください)。

7.6.1 while 文

while文は、ループ本体の各実行の前に制御式を評価します。制御式が真 (0 以外) の場合はループ本体が実行され、偽 (0) の場合は while文は終了します。while文の構文は次のとおりです。

```
while ( 式 )
```

文

次に while文の例を示します。

```
n = 0;
while (n < 10)
{
    a[n] = n;
    n++;
}
```

この文は n の値をチェックします。 n が 10 未満の場合は、 a 配列の n 番目の要素に n を割り当てて、 n を増分します。次に、(括弧内の) 制御式を評価します。真 (0 以外) の場合はループ本体が再び実行され、偽 (0) の場合は while文は終了します。 $n++$ がループ本体にない場合には、この while文は終了することはありません。この例の $n = 0$ の文を $n = 10$ の文に置き換えると、この制御式は最初から偽 (0) になり、ループ本体が実行されることはありません。

7.6.2 do 文

do文は、ループ本体の各実行の後に制御式を評価します。

do文の構文は次のとおりです。

```
do
```

文

```
while ;
```

文

7.6 繰返し文

ループ本体は少なくとも 1 回は実行されます。制御式は、ループ本体の各実行の後に評価されます。制御式が真 (0 以外) の場合は文が再び実行され、偽 (0) の場合は do 文は終了します。

7.6.3 for 文

for 文は 3 つの式を評価し、2 番目の制御式の評価が偽 (0) になるまでループ本体を実行します。for 文は、ループ本体を指定回数だけ実行する場合に役に立ちます。for 文の構文は次のとおりです。

```
for ( 式 1opt ;  
      式 2opt ; 式 3opt )  
  文
```

for 文は次のコードと等価です。

```
式 1 ;  
while ( 式 2 )  
{  
  文  
  式 3 ;  
}
```

for 文はループ本体を 0 回以上実行します。セミコロン (;) を使用して、制御式を区切ります。for 文は次の処理を行います。

1. 式 1 は、ループの最初の繰返しの前に 1 回評価されます。この式は通常、ループ中で使用する変数の初期値を指定します。
2. 式 2 は、ループを終了するかどうかを決めるスカラ式です。式 2 は各ループの繰返しの前に評価されます。式が真 (0 以外) の場合はループ本体が実行され、偽 (0) の場合は for 文の実行は終了します。
3. 式 3 は各繰返しの後に評価されます。
4. for 文は式 2 が偽 (0) になるまで、あるいは break または goto などの飛越し文でループの実行を終了するまで実行されます。

forループ内の3つの式はいずれも省略できます。次にその例を示します。

- 式2を省略すると、判定条件は常に真になります。すなわち、whileループの等価は while(1) になります。これは無限ループです。次にその例を示します。

```
for (i = 0; ; i++)
    文;
```

無限ループはループ本体内の break, return, または goto文で終了させることができます。

- 式1または式3を for文から省略すると、省略した式は void式として評価され、実際の展開からは除外されます。次にその例を示します。

```
n = 1;
for ( ; n < 10; n++)
    func(n);
```

この例では、n は for文が実行される前に初期化されます。

relaxed ANSI C モードでは、for文の最初の句で、forヘッダの残りの句とループ本体をスコープとする宣言を行うことができます。通常この句は、ローカルなループ制御変数の宣言と初期化に使用します。次の例を参照してください。

```
for (int i=0; i<10; i++)
    printf("%d\n", i);
```

7.7 飛越し文

飛越し文は、コード内の別の場所にある別の文への無条件飛越しを引き起こします。飛越し文は主に、switch文およびループの割込みに使用されます。

飛越し文とは、goto文、continue文、break文、および return文です。これ以後の各項では、これらの文について説明します。

文 7.7 飛越し文

7.7.1 goto 文

ラベル識別子とその goto文を含む関数のスコープにある場合，goto文はプログラム制御をそのラベル付き文に無条件に渡します。次に，制御を渡されたラベル付き文が実行されます。goto文の構文は次のとおりです。

goto 識別子 ;

goto文を使用してブロックに分岐する際には，注意する必要があります。つまり，ブロックの呼出し時には，そのブロック内で宣言された自動変数に対して記憶域が割り当てられるのに対して，goto文でブロックに分岐すると，そのブロック内で宣言された自動変数は初期化されないからです。

7.7.2 continue 文

continue文は，それを直接囲んでいる while，do，または for文のループの終わりに制御を渡します。continue文の構文は次のとおりです。

continue ;

continue文は，ループ本体の終わりに制御を渡す繰返し文の中の goto文と同じです。たとえば，次の2つのループは同じです。

while(1)	while(1)
{	{
.	.
.	.
.	.
goto label_1;	continue;
.	.
.	.
.	.
label_1:	
;	;
}	}

continue文はループ内でのみ使用できます。ループの内部にあるswitch文にcontinue文がある場合には，continue文は switch文の本体が終了し，外側のループの処理を続行します。

7.7.3 break 文

break文は、それを直接囲んでいる while, do, または switch文の実行を終了します。制御はループ本体に続く文 (または switch文の複文) へ渡されます。break文の構文は次のとおりです。

```
break ;
```

switch文の中での break文の使用例については、例 7-1 を参照してください。

7.7.4 return 文

return文は関数の実行を終了し、制御を返却値とともにまたは返却値なしで呼出し関数に返します。関数に含まれる return文の数に制限はありません。

return文の構文は次のとおりです。

```
return 式opt ;
```

式が存在する場合はその式は評価され、その値が呼出し関数へ返されます。その値は必要に応じて、それを含む関数の返却値に宣言されている型へ変換されます。

式を持つ return文は、返却値の型が void の関数には指定することはできません。voidデータ型および関数の返却値の型についての詳細は、第 3.5 節 と第 3.4.1 項を参照してください。

式が存在しない場合で関数が void として定義されていなければ、返却値の型も定義されていません。たとえば、次の main関数は予測できない値をオペレーティング・システムに返します。

```
main ( )
{
    return;
}
```

関数を終了する右中括弧に達すると、式を持たない return文を実行することになります。

前処理命令とあらかじめ定義されたマクロ

C プリプロセッサは、マクロ置換、条件付きコンパイル、および指定したファイルの取込みを実行する機能を提供します。前処理命令はプリプロセッサとの連絡に使用する文であり、その先頭には # を付けます。# の前に空白が存在する場合もあります。

この章の各節では、*Compaq C* コンパイラで使用可能な前処理命令および演算子について説明します。

- #define および #undef 命令と # および ## 演算子 (第 8.1 節)
- #if, #ifdef, #ifndef, #else, #elif, および #endif 命令と defined 演算子 (第 8.2 節)
- #include 命令 (第 8.3 節)
- #line 命令 (第 8.4 節)
- #pragma 命令 (第 8.5 節)
- #error 命令 (第 8.6 節)
- 空命令 (#) (第 8.7 節)

前処理命令は通常のスコープの規則とは独立しており、コンパイル単位の終わり、またはその効力が取り消されるまで有効です。

条件付きコンパイルについての詳細は、第 8.2 節を参照してください。前処理命令の処理系に依存する定義についての詳細は、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

ANSI 規格では、前処理命令に続けるテキストとしてコメントだけを許可しています。厳密な ANSI モード以外のすべてのモードでこの構文規則に違反した場

合, *Compaq C*コンパイラは警告を出します。厳密な ANSI モードで違反した場合には, エラー・メッセージを出します。

8.1 マクロ定義 (#define および #undef)

#define 命令はマクロ識別子と置換並びを指定し, 改行文字で終了します。置換並び, つまり一連の前処理トークンはプログラム・テキスト内でマクロ識別子が出現するたびに置き換えられます。ただし, このマクロ識別子が文字定数, コメント, またはリテラル文字列の内部で出現した場合には置き換えられません。#undef 命令は, マクロ定義を取り消す場合に使用されます。

マクロ定義はブロック構造からは独立しており, このマクロを定義している #define 命令から, それに対応する #undef 命令またはコンパイル単位の終わりが現れるまで有効です。

#define 命令の構文は次のとおりです。

```
#define 識別子 置換並び 改行
#define 識別子 ( 識別子並びopt ) 置換並び 改行
#define 識別子 ( ... ) 置換並び 改行
#define 識別子 ( 識別子並び, ... ) 置換並び 改行
```

置換並びが空の場合, その識別子がそれ以降に出現するとソース・ファイルから削除されます。

#define 命令の 1 番目の形式をオブジェクト形式マクロと呼びます。残りの 3 つの形式を, 関数形式マクロと呼びます。

#undef 命令の構文は次のとおりです。

```
#undef 識別子 改行
```

この命令は, #define によって定義された以前の識別子の定義を取り消します。以前に定義されたマクロの再定義は, 新しい定義が古い定義と厳密に同じではない限り無効になります。

マクロ定義内の置換並びには、関数形式のマクロ参照の実引数と同様に他のマクロ参照を含めることができます。 *Compaq C* では、これらの参照のネストされた深さを制限していません。

所定のマクロ定義により、置換並びに入っているそのマクロ名自体を現在指定している置換並びに置き換えることができます。ただし、定義しているマクロ名がそのマクロ名自体の置換並び、またはネストされた置換並びに入っている場合にはそのマクロ名は置き換えられません。この置き換えられなかったマクロ名は後で別の置換が行われる文脈内で検査されても、その置換には使用できなくなります。

次の例は、ネストされた #define命令を示しています。

```
/* Show multiple substitutions and listing format. */
#define AUTHOR james + LAST
main()
{
    int writer,james,michener,joyce;

    #define LAST michener
    writer = AUTHOR;
    #undef LAST
    #define LAST joyce
    writer = AUTHOR;
}
```

この例を、中間マクロ展開を表示させるための適切なオプションでコンパイルすると、次の並びが生成されます。

```
1          /* Show multiple substitutions and listing format. */
2
3          #define AUTHOR james + LAST
4
5          main()
6          {
7              int writer, james, michener, joyce;
8
9              #define LAST michener
10             writer = AUTHOR;
10.1             james + LAST
10.2             michener
```

前処理命令とあらかじめ定義されたマクロ

8.1 マクロ定義 (#defineおよび#undef)

```
11          #undef LAST
12          #define LAST joyce
13          writer = AUTHOR;
13.1              james + LAST
13.2              joyce
14      }
```

1 回目のパスでは、コンパイラは AUTHOR 識別子を james + LAST 置換並びに置き換えます。2 回目のパスでは、コンパイラは LAST 識別子を現在定義されている置換並び値に置き換えます。9 行目で LAST の置換並び値は michener 識別子になるので、10 行目で michener に置き換えられます。12 行目で LAST の置換並び値は joyce 識別子に再定義されるので、13 行目で joyce に置き換えられます。

#define 命令を必要に応じて次の行に継続させることができます。このためには、次に継続行が来る行をバックスラッシュ (\) で終了し、その直後に改行文字を入力します。バックスラッシュと改行文字は定義の一部にはなりません。次の行の先頭文字は、論理的にはバックスラッシュの直前の文字に隣接します。継続シーケンスとしてのバックスラッシュおよび改行文字は、どこでも有効です。ただし、定義行にあるコメントはバックスラッシュまたは改行文字を指定せずに継続させることができます。

他の C 処理系との間でプログラムを移植する場合は処理系によって定義するマクロが異なるため、プログラム内で使用するマクロを選択する際には注意が必要です。

8.1.1 オブジェクト形式マクロ

次の形式の前処理命令はオブジェクト形式マクロを定義します。これによって、マクロ名はそれ以後に出現するたびに置換並びに置き換えられます。

```
#define 識別子 置換並び 改行
```

オブジェクト形式マクロは、別の #define 命令によって再定義することができます。ただし、この場合は 2 番目の定義がオブジェクト形式マクロであり、2 つの置換並びが同一でなければなりません。つまり、2 つのファイルにある特定のマクロ定義は整合するものでなければなりません。

オブジェクト形式のマクロ定義は、頻繁に使用されるトークンの記述名を定義します。一般に、命令を使用してファイルの終わり (EOF) の指示子を次のように定義します。

```
#define EOF (-1)
```

8.1.2 関数形式マクロ

関数形式のマクロ定義は、仮引数並びを含んでいます。このマクロの参照は関数呼出しに類似しています。関数を呼び出すと、実行時にプログラムから関数へ制御が渡されます。マクロが参照されると、ソース・コードがコンパイル時にプログラムへ挿入されます。仮引数は対応する実引数に置き換えられ、テキストはプログラム・ストリームへ挿入されます。

マクロ定義の識別子並びに... がある場合、末尾の引数 (区切りのカンマ前処理トークンを含む) は、1つの項目 (可変長引数) を形成するようにマージされます。このマージによって引数が結合された後の引数の数は、マクロ定義内のパラメータの数 (... を除く) より1大きくなります。

反復記号表記を引数内で使用している関数形式マクロの置換並びにある `__VA_ARGS__` 識別子は、パラメータのように扱われます。また、可変長引数により、この識別子を置き換えるための前処理トークンが形成されます。

置換並びをマクロ定義から省略すると、そのマクロ参照全体がソース・テキストから消えます。

マクロ置換の例として、一部のシステム上の `ctype.h` ヘッダ・ファイル内で使用可能な `_toupper` ライブラリ・マクロを示します。このマクロは次のように定義します。

```
#define _toupper(c) ((c) >= 'a' && (c) <= 'z' ? (c) & 0X5F : (c))
```

`_toupper` マクロが参照されると、コンパイラはマクロとその仮引数をこの命令の置換並びに置き換え、置換並びにある仮引数 (この例では `c`) が出現するたびにマクロ参照の実引数を置き換えます。

前処理命令とあらかじめ定義されたマクロ

8.1 マクロ定義 (#defineおよび#undef)

この C 言語のソース・コードの置換並びは次のように翻訳されます。すなわち，c 仮引数が小文字 ('a' から 'z' まで) の場合は式を大文字 (c & 0X5F) として評価し，それ以外の場合は式を指定どおりの文字として評価します。この置換並びでは，if-then-else 条件演算子 (?:) を使用しています。条件演算子についての詳細は，第 6.6 節を参照してください。ビット演算子についての詳細は，第 6.5.6 項を参照してください。

8.1.2.1 マクロ定義指定の規則

前処理命令とマクロ参照は，C 言語からは独立した構文を持っています。マクロ定義の指定時には次の規則に従ってください。

- マクロ名と仮引数は識別子であり，*Compaq C* の識別子の規則に従って指定する。
- #define 命令では，空白，タブ，およびコメントを使用できる。次の例のデルタ記号 (Δ) で示した場所のどこにでも使用できる。

```
Δ #Δ define Δ 名前 (Δ 仮引数1Δ , Δ 仮引数2Δ ) Δ \
```

```
Δ トークン文字列Δ
```

空白，タブ，およびコメントは 1 つの空白に置き換えられる。

- 名前と仮引数並びを開始する左括弧の間には，空白を入れることはできない。置換並びの中には空白を入れることができない。また，define と名前の間は少なくとも 1 つの空白，タブ，またはコメントで区切らなければならない。
- __VA_ARGS__ 識別子は，引数内で反復記号表記を使用している関数形式マクロの置換並びでのみ使用できる。

8.1.2.2 マクロ参照指定の規則

マクロ参照の指定時には次の規則に従ってください。

- マクロ参照では，コメントおよび空白文字 (空白，水平タブ，垂直タブ，改行文字，およびフォーム・フィード) を使用することができる。次の例のデルタ記号 (Δ) で示した場所のどこにでも使用できる。

```
Δ 名前Δ (Δ 実引数1Δ , Δ 実引数2Δ )
```

- 実引数は任意のテキストから構成される。構文上からは実引数はC言語の式に制限されず、埋め込みコメントや空白を含めることができる。コメントは1つの空白に置き換えられる。空白(先行および後続の空白を除く)は置換時に保存される。
- 参照中の実引数の数は、マクロ定義中の仮引数の数と一致しなければならない。空実引数の場合の動作結果は定義されていない。
- コンマは、文字列や文字定数、コメント、または括弧の内部に使用される場合を除き、実引数を区切る際に使用される。括弧は実引数の中で対になっていないなければならない。

8.1.2.3 マクロ実引数における副作用

インクリメント演算子(++)、デクリメント演算子(--), 代入演算子(+= など), またはその他の副作用を起こす可能性がある実引数を使用するマクロ実引数を指定することは、正しいプログラミング技法ではありません。たとえば、次の実引数は_toupperマクロに渡してはなりません。

```
_toupper(p++)
```

p++実引数がマクロ定義内で置き換えられると、プログラム・ストリーム内では次のようになります。

```
((p++) >= 'a' && (p++) <= 'z' ? (p++) & 0X5F : (p++))
```

pは増分されるため、このマクロ置換で出現するたびに異なる値を持ちます。副作用が起こる可能性を認識していても、マクロ定義内の置換並びは変更されることがあります。さらに、この変更によって副作用が変わっても警告は出されません。

8.1.3 文字列リテラルへの変換(#)

#前処理演算子は、次に続く実引数を文字列リテラルに変換する場合に使用します。#前処理演算子は、関数形式マクロ定義でのみ使用することができます。次にその例を示します。

前処理命令とあらかじめ定義されたマクロ

8.1 マクロ定義 (#defineおよび#undef)

```
#include <stdio.h>

#define PR(id) printf("The value of " #id " is %d\n", id)

main()
{
    int i = 10;

    PR(i);
}
```

生成される出力は、次のとおりです。

```
The value of i is 10
```

マクロ呼出しは、次のように展開されます。

```
/*1*/ printf("The value of " #id " is %d\n", id)
/*2*/ printf("The value of " "i" " is %d\n", 10)
/*3*/ printf("The value of i is %d\n", 10)
```

#単項演算子は、オペランドから文字列を生成します。この例では、隣接する文字列リテラルが連結されるという事実も示しています。#のオペランドに二重引用符またはエスケープ・シーケンスが含まれている場合には、これらも展開されます。次にその例を示します。

```
#include <stdio.h>

#define M(arg) printf(#arg "is %s\n", arg)

main()
{
    M("a\nb\tc");
}
```

マクロ呼出しは、次のように展開されます。

```
/*1*/ printf(#arg "is %s\n", arg)
/*2*/ printf("\"a\nb\tc\" " "is %s\n", "a\nb\tc");
/*3*/ printf("\"a\nb\tc\" is %s\n", "a\nb\tc");
```

8.1.4 トークンの連結 (##)

##前処理演算子は、2つのトークンを連結して1つの有効なトークンにする場合に使用します。次にその例を示します。

```
#define glue(a,b) a ## b

main()
{
    int wholenum = 5000;
    printf("%d", glue(whole,num));
}
```

プリプロセッサは、`printf("%d", glue(whole,num));` の行を `printf("%d", wholenum);` に変換します。これが実行されると、プログラムは5000を表示します。2つのトークンを連結したトークンが有効ではない場合には、エラーが生じます。

*Compaq C*では、##演算子はその行にある#演算子の前に評価されます。## および#演算子は、左から右へグループ分けされます。

8.2 条件付きコンパイル (#if , #ifdef , #ifndef , #else , #elif , #endif , defined)

条件付きコンパイルの制御には、6つの命令を使用することができます。これらの命令は、指定した条件が真の場合にのみコンパイルされるプログラム・テキスト・ブロックを区切ります。これらの命令はネストすることができます。ブロック内のプログラム・テキストは任意に指定でき、前処理命令、C文などで構成することができます。プログラム・テキスト・ブロックの先頭には、次の3つの命令のいずれかを指定します。

- #if
- #ifdef
- #ifndef

オプションとして、次の 2 つの命令のいずれかで代替テキスト・ブロックをセットすることができます。

- #else
- #elif

ブロックまたは代替ブロックの終わりには、#endif命令を指定します。

#if, #ifdef, または #ifndef で判定した条件が真 (0 以外) の場合には、それに対応する #else (または #elif) 命令から #endif命令までの行 (存在する場合) は無視されます。

条件が偽 (0) の場合には、#if, #ifdef, または #ifndef命令から #else, #elif, または #endif命令までの行が無視されます。

8.2.1 #if 命令

#if命令の形式は次のとおりです。

#if 定数式 改行

この命令は定数式が真 (0 以外) かどうかを判定します。オペランドはインクリメント演算子 (++)、デクリメント演算子 (--), sizeof演算子、ポインタ演算子 (*), アドレス演算子 (&), およびキャスト演算子を含んでいない定数整数式でなければなりません。

定数式中の識別子はマクロ名の場合もあり、そうでない場合もあります。キーワードや列挙定数などは含みません。定数式には、defined前処理演算子を含めることもできます (第 8.2.7 項を参照してください)。

#if命令中の定数式はテキスト置換の対象となり、#define命令で前に定義された識別子への参照を含めることができます。置換は式が評価される前に行われます。すべてのマクロ置換が行われた後に残った前処理トークンは、字句解析形式のトークンになります。

式中で使用した識別子が現在定義されていない場合には、コンパイラはこの識別子を定数 0 として処理します。

8.2.2 #ifdef 命令

#ifdef命令の構文は以下のとおりです。

#ifdef 識別子 改行

この命令は識別子が現在定義されているかどうかを検査します。識別子は、
#define命令またはコマンド行で定義することができます。定義された識別子はそ
の後定義が取り消されない限り、現在定義済みであるとみなされます。

8.2.3 #ifndef 命令

#ifndef命令の構文は以下のとおりです。

#ifndef 識別子 改行

この命令は識別子が現在定義されていないかどうかを検査します。

8.2.4 #else 命令

#else命令の構文は以下のとおりです。

#else 改行

この命令は、対応する #if, #ifdef, または #ifndef命令で判定された条件が偽の
場合にコンパイルする代替ソース・テキストを区別します。#else命令はオプショ
ンです。

8.2.5 #elif 命令

#elif命令の構文は以下のとおりです。

#elif 定数式 改行

#elif命令は、C 言語の else-if文の複合使用と類似したタスクを実行します。こ
の命令は、対応する #if, #ifdef, #ifndef, または別の #elif命令中の定数式が偽
であり、さらに #elif行の定数式が真である場合にコンパイルする代替ソース行を
区別します。#elif命令はオプションです。

前処理命令とあらかじめ定義されたマクロ

8.2 条件付きコンパイル (#if, #ifdef, #ifndef, #else, #elif, #endif, defined)

8.2.6 #endif 命令

#endif命令の構文は以下のとおりです。

#endif 改行

この命令は #if, #ifdef, #ifndef, #else, または #elif命令のスコープを終了します。

必要な #endif命令の数は, #elif または #else命令のいずれが使用されているかによって異なります。次に 2 種類の例を示します。

#if true	#if true
.	.
.	.
.	.
#elif true	.
.	#else
.	#if false
.	.
#endif	.
	.
	#endif
	#endif

8.2.7 defined 演算子

マクロが定義済みであることを確認する別の方法は, defined単項演算子を使用する方法です。defined演算子の形式は次のいずれかです。

defined 名前

defined (名前)

この形式の式は名前が定義済みの場合には 1 と評価され, 定義済みではない場合には 0 と評価されます。

defined演算子は, #if命令を 1 回だけ使用して複数のマクロを検査する場合に特に役に立ちます。この方法を使用すれば, 多くの #ifdef または #ifndef命令を使用せずに, 簡潔な 1 行だけでマクロ定義を検査することができます。

次にマクロ検査の例を示します。

```
#ifdef macro1
printf( "Hello!\n" );
#endif

#ifndef macro2
printf( "Hello!\n" );
#endif

#ifdef macro3
printf("Hello!\n");
#endif
```

これと同様のマクロ検査を行うために、defined演算子を単一の #if命令中に指定する方法があります。次にその例を示します。

```
#if defined (macro1) || !defined (macro2) || defined (macro3)
printf( "Hello!\n" );
#endif
```

*Compaq C*の論理演算子を使用すると、defined演算子を論理式の中で組み合わせることができます。ただし、defined は #if または #elif前処理命令の評価済みの式でのみ使用できます。

8.3 ファイルの取込み (#include)

#include命令は、指定したファイルの内容をコンパイラへ引き渡されるテキスト・ストリームへ挿入します。標準ヘッダおよびグローバル定義は通常、#include命令でプログラム・ストリームに取り込まれます。#include命令の形式には、次の2種類があります。

```
#include "ファイル名" 改行
#include <ファイル名> 改行
```

ファイル名の形式は、各プラットフォームに依存します。ファイル名が引用符で囲まれている場合、指定されたファイルの検索は#include命令を含むファイルが存在しているディレクトリから始まります。ファイルがこのディレクトリにない場合、またはファイル名が山括弧(< >)で囲まれている場合には、ファイルの検索は

各プラットフォームに定義されている検索規則に従います。ファイル名が引用符で囲まれている形式の #include は一般的に、ユーザ作成のファイルを取り込む場合に使用され、山括弧形式の #include は標準ライブラリ・ファイルを取り込む場合に使用されます。

ファイルの取込みのための検索パスの規則についての詳細は、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

マクロ置換は、#include 前処理命令の中で使用できます。

たとえば、次の 2 つの命令を使用してファイルを取り込むことができます。

```
#define macrol "file.ext"
#include macrol
```

#include 命令の中で使用した定義済みマクロは、次の 2 つの受け入れ可能な #include ファイル指定の一方に評価されなければなりません。評価できない場合にはエラーが報告されます。

"ファイル名"
<ファイル名>

取り込まれるファイル自体に #include 命令を含めることができます。 *Compaq C* コンパイラには、取込みのネストされた深さに固有の制限はありませんが、指定できる深さは使用するハードウェアおよびオペレーティング・システムの制約によって異なります。

8.4 明示的な行番号付け (#line)

コンパイラはコンパイルで使用了各ファイルの行番号に関する情報を追跡し、ターミナルに診断メッセージを表示する際、またはバッチ・モードでのコンパイル時にログ・ファイルに診断メッセージを記録する際に、行番号を使用します。

#line 命令を使用すると、ソース・コードに割り当てた行番号を変更することができます。この命令は次の行に新しい行番号を与え、それ以降の行にそれに続く行番号を生成します。この命令は、プログラム・ソース・ファイルに対して新しいファ

イル指定を指定することもできます。#line命令はコンパイル並び中の行番号は変更せずに、ターミナル画面またはログ・ファイルへ送られる診断メッセージの中で与えた行番号だけを変更します。この命令は、Cコードへ前処理された元のソース・ファイルを参照する場合に役に立ちます。

#line命令の形式には、次の3種類があります。

#line 整数定数 改行

#line 整数定数 "ファイル名" 改行

#line 前処理トークン 改行

最初の2つの形式では、コンパイラは#line命令に従って、整数定数が指定した番号を行に与えます。引用符の中のオプションのファイル名は、コンパイラが診断メッセージの中で提供するソース・ファイルの名前を示します。ファイル名を省略した場合には、現在のソース・ファイルの名前、または前の#line命令の中で指定された最後のファイル名が使用されます。

3番目の形式では、#line命令の中のマクロは展開された後に解釈されます。これによって、マクロ呼出しは整数定数、ファイル名、またはその両方に展開することができます。その結果生成された#line命令は、他の2つの形式のいずれかに一致しなければなりません。その後、必要に応じて処理が行われます。

8.5 処理系固有の前処理命令 (#pragma)

#pragma命令は、各プラットフォームに依存する機能を実行するための標準の方法です。この命令の構文は以下のとおりです。

#pragma 前処理トークン_{opt} 改行

サポートされるプリAGMAは、プラットフォームによって異なります。認識されないプリAGMAにはすべて情報メッセージが出されます。サポートされるプリAGMAの一覧については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

前処理命令とあらかじめ定義されたマクロ

8.5 処理系固有の前処理命令 (#pragma)

次のプリAGMA命令は、マクロ展開が行われます。

<code>builtin</code>	<code>inline</code>	<code>linkage</code>	<code>standard</code>
<code>dictionary</code>	<code>noinline</code>	<code>module</code>	<code>nostandard</code>
<code>extern_model</code>	<code>member_alignment</code>	<code>message</code>	<code>use_linkage</code>
<code>extern_prefix</code>	<code>nomember_alignment</code>		

次のプリAGMAでもマクロ展開が行われますが、これらは主にプリプロセッサ専用モード (つまり、*OpenVMS* システムでの `/PREPROCESS_ONLY` 修飾子、または、*Tru64 UNIX* システムでの `-E` スイッチの使用) で使用するためのものであり、*Compaq C* コンパイラでオブジェクト・モジュールを生成する際には、通常は使用されません。

- `_KAP`—*KAPC* 製品にのみ関連する。
- `define_template`—*Compaq C++* にのみ関連する。
- `code_psect`
- `linkage_psect`

注意

上記のマクロのいずれにも、`_nm` 接尾語を付加してマクロ展開を抑制することができます。たとえば、`#pragma inline` でのマクロ展開を抑制するには、`#pragma inline_nm` のように指定します。

また、上記のリストにないプリAGMAのマクロ展開をサポートするために、すべてのプリAGMA (マクロ展開対象としてすでに指定されているものも含む) には、マクロ展開の対象となる `pragma-name_m` バージョンがあります。たとえば、`#pragma assert` はマクロ展開の対象になりませんが、`#pragma assert_m` はマクロ展開の対象になります。

マクロ参照は、`pragma` キーワードの後ろの任意の位置に記述できます。例を次に示します。

```
#define opt inline
#define f func
#pragma opt(f)
```

両方のマクロが展開されると、この#pragma命令は#pragma inline (func)になります。

注意

マクロ展開はCompaq Cの初期のバージョンに導入されたプラグマの特徴であり、下位互換性のために保持されています。

最近のバージョンのコンパイラに追加されているプラグマと、将来のバージョンに追加されるプラグマは、マクロ展開を行わないという事実上の業界標準に準拠するように変更されています。ANSI Cでは、プラグマのマクロ展開について何の要件も規定していません。

次に、コンパイラが、任意のプラグマのマクロ展開を行うかどうかをどのように決定するかについて説明します。

コンパイル・モードが/STANDARD=COMMON (*OpenVMS*システム) または -std0 (*Tru64 UNIX*システム) 以外の場合は、ステップ1を行います。

ステップ1:

まず、キーワードpragmaに続くトークンがチェックされて、最近定義されたマクロであるかどうか判断される。それがマクロであって、識別子がマクロ展開を行わないプラグマの名前と一致しない場合には、そのマクロ (関数形式の場合はその仮引数も) だけが展開される。そのマクロ展開によって作成されたトークンは、ステップ2で同じ行にある残りのトークンと一緒にそのまま処理される。

すべてのコンパイル・モードで、ステップ2を行います。

ステップ2

キーワードpragmaに続く最初のトークンがチェックされて、マクロ展開を行うプラグマの名前と一致するかどうか判断される。一致した場合には、そのトークンと同じ行の残りのトークンに対してマクロ展開が適用される。

前処理命令とあらかじめ定義されたマクロ

8.5 処理系固有の前処理命令 (#pragma)

既知のプリAGMAに一致するかどうかのテストでは、先頭に2つのアンダスコア(`_`)が付いていてもかまいません。たとえば、`#pragma __nostandard`は`#pragma nostandard`と同等です。

例

次の例は、既知のプリAGMAに一致する名前で直接コーディングされているプリAGMAについては、一般にマクロ展開の動作はすべてのモードで同じであり、下位互換性があることを示しています。下位互換性がないのは、プリAGMAの名前に既知のプリAGMAの名前以外を使用してコーディングし、マクロ展開をしてプリAGMA名を生成することを予期している場合だけで、これは共通モードでだけ起こります。この例外は、*Tru64 UNIX*プリプロセッサとの互換性を維持するために、共通モードで設けられています。

```
#define pointer_size error
#define m1 e1
#define e1 pointer_size 32
#define standard message
#define x disable(all)
#define disable(y) enable(y)

#pragma pointer_size 32 /* In common mode, Step 1 skipped.
                        In other modes, Step 1 finds that pointer_size
                        is known not to expand.
                        In any mode, Step 2 finds pointer_size is
                        not a pragma requiring expansion. */

#pragma m1 /* In common mode, Step 1 skipped.
            In other modes, Step 1 expands m1 to pointer_size 32.
            In common mode, Step 2 finds m1 is not a pragma requiring
            expansion.
            In other modes, Step 2 finds pointer_size is not a pragma
            requiring expansion. */

#pragma standard x /* In common mode, Step 1 skipped.
                    In other modes, Step 1 expands to message x.
                    In common mode, Step 2 expands to message enable(all).
                    In other modes, Step 2 expands message x to
                    message enable(all). */
```

8.6 エラー命令 (#error)

#error前処理命令は、E レベルの診断メッセージを表示してコンパイルを続行しますが、オブジェクト・モジュールは生成しません。この命令の構文は次のとおりです。

#error メッセージ_{opt} 改行

8.7 空命令 (#)

#改行 形式の前処理命令は空命令であり、何の処理も行われません。

8.8 あらかじめ定義されたマクロ名

これ以降の項ではあらかじめ定義されたマクロについて説明します。これらのマクロは、コードの移植および多くのプログラムに共通する単純なタスクの実行を支援するために提供されています。

8.8.1 __DATE__マクロ

__DATE__マクロは、コンパイル開始日を示す文字列リテラルを表します。形式は次のとおりです。

"Mmm dd yyyy"

月の名前は、asctimeライブラリ関数によって生成されるものと同じです。dd が10未満の場合には、最初のdは空白です。次にその例を示します。

```
printf("%s", __DATE__);
```

このマクロの値は1つの翻訳単位内では一定です。

8.8.2 __FILE__マクロ

__FILE__マクロは、現在のソース・ファイルのファイル指定を指定する文字列リテラルを表します。次にその例を示します。

```
printf("file %s", __FILE_);
```

8.8.3 __LINE__マクロ

__LINE__マクロは、そのマクロ参照を含むソース・ファイルの行番号を指定する10進定数を表します。次にその例を示します。

```
printf("At line %d in file %s", __LINE_, __FILE_);
```

8.8.4 __TIME__マクロ

__TIME__マクロは、コンパイルの開始時刻を示す文字列を表します。時刻の形式は次のとおりです (asctime関数と同様です)。

```
hh:mm:ss
```

次にその例を示します。

```
printf("%s", __TIME_);
```

このマクロの値は1つの翻訳単位内では一定です。

8.8.5 __STDC__マクロ

__STDC__マクロは、規格合致の処理系であることを示す整数定数1を表します。

このマクロの値は1つの翻訳単位内では一定です。

8.8.6 __STDC_HOSTED__ マクロ

__STDC_HOSTED__ マクロは、ホスト処理系の場合には整数定数 1 と評価され、ホスト処理系でない場合には整数定数 0 と評価されます。

8.8.7 __STDC_VERSION__ マクロ

__STDC_VERSION__ マクロは、整数定数 199901L と評価されます。

8.8.8 __STDC_ISO_10646__ マクロ

__STDC_ISO_10646__ マクロは、yyyymmL (たとえば、199712L) の形式の整数定数と評価されます。このマクロは、wchar_t 型の値が、ISO/IEC 10646 (指定された年月までのすべての改正 (amendment) および技術的訂正 (technical corrigenda) を含む) で定義されている文字のコード化表現であることを示します。

8.8.9 システム識別マクロ

Compaq C では、プログラムを実行しているシステムを識別するために使用するプラットフォーム固有のマクロを定義することができます。これらのマクロは、プログラムを DEC のシステムまたは他社のシステムのいずれで実行するのか、あるいはどの *Compaq C* プラットフォームで実行するのかに応じて、条件付きで実行するコードを記述する際に役に立ちます。

これらのマクロ定義を使用すれば、移植不可能なコードを条件付きコンパイル・セクションに囲い込むことによって、C プログラムにおける移植可能なコードと移植不可能なコードを分離することができます。

さらに、これらのマクロ定義を使用して、複数のオペレーティング・システムで使用される C プログラムのセクションを条件付きでコンパイルし、システム固有の機能を有効に利用することもできます。条件付きコンパイルの前処理命令の使用についての詳細は、第 8.2 節を参照してください。

システム識別マクロの一覧については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

8.9 __func__ 宣言済み識別子

`__func__` 宣言済み識別子は、関数名の綴りで初期化された、`char` 型の静的配列として評価されます。この識別子は、関数定義本体内のどこからでも認識できます。

たとえば、次のように定義された関数は、`"f1"` を出力します。

```
void f1(void) {printf("%s\n", __func__);}
```

ANSI C 標準ライブラリ

ANSI C 規格は、ANSI C の処理系で提供される一連の関数とそれに関連する型およびマクロを定義しています。この章では、すべての *Compaq C* プラットフォームに共通の ANSI 規格に準拠するライブラリ機能について簡単に説明します。*Compaq C* ライブラリ・ルーチンとそのシステム環境での使用、および各オペレーティング・システムで使用可能な追加のヘッダ、関数、型、およびマクロについての詳細は、該当する *Compaq C* ライブラリ・ルーチンのマニュアルを参照してください。

すべてのライブラリ関数はヘッダ・ファイルに宣言されています。ヘッダ・ファイルの内容をプログラムで使用可能にするには、そのヘッダ・ファイルを `#include` 前処理命令で取り込みます。次にその例を示します。

```
#include <stddef.h>
```

各ヘッダ・ファイルは一連の関連する関数を宣言し、これらを使用するために必要な型とマクロを定義します。

標準ヘッダは次のとおりです。

- 診断メッセージ: `<assert.h>` (第 9.1 節)
- 複素数算術: `<complex.h>` (第 9.2 節)
- 文字処理: `<ctype.h>` (第 9.3 節)
- エラー・コード: `<errno.h>` (第 9.4 節)
- ANSI C の限界: `<limits.h>` および `<float.h>` (第 9.5 節)
- ローカル化: `<locale.h>` (第 9.6 節)
- 算術: `<math.h>` (第 9.7 節)
- 非ローカル飛越し: `<setjmp.h>` (第 9.8 節)

- シグナル処理: <signal.h> (第 9.9 節)
- 可変個実引数: <stdarg.h> (第 9.10 節)
- ブール型とブール値: <stdbool.h> (第 9.11 節)
- 共通定義: <stddef.h> (第 9.12 節)
- 入出力: <stdio.h> (第 9.13 節)
- 汎用ユーティリティ: <stdlib.h> (第 9.14 節)
- 文字列処理: <string.h> (第 9.15 節)
- 型汎用数学: <tgmath.h> (第 9.16 節)
- 日付と時刻: <time.h> (第 9.17 節)

ヘッダ・ファイルは、どのような順序でも取り込むことができます。各ヘッダ・ファイルは所定のスコープ内で複数回取り込むことができ、これは 1 回取り込んだ場合と同じ結果になります。ただし、<assert.h> を取り込んだ場合の結果は NDEBUG の定義によって異なります。ヘッダの取込みは外部宣言または定義の外側で、そのヘッダに宣言または定義されている関数、型、またはマクロを参照する前に行います。識別子が複数の取込みヘッダの中で宣言または定義されている場合、この識別子を含む 2 番目以降のヘッダはこの識別子への最初の参照の後に取り込むことができます。

9.1 診断メッセージ (<assert.h>)

<assert.h> ヘッダは assert マクロを定義し、別の場所で定義された別の NDEBUG マクロを参照します。ソース・ファイル内の <assert.h> が取り込まれた場所で NDEBUG がマクロ名として定義されている場合には、assert マクロは次のように定義されます。

```
#define assert(ignore) ((void) 0)
```

マクロ

```
void assert(int expression);
```

診断メッセージをプログラムに含める。式が偽 (0) の場合，assertマクロは失敗した特定の呼出しに関する情報を標準エラー・ファイルに処理系定義の書式で書き込む。次に，マクロは abort関数を呼び出す。assertマクロは値を返さない。

9.2 複素数算術 (<complex.h>)

<complex.h>ヘッダ・ファイルは，マクロを定義し，複素数算術をサポートする関数を宣言しています。

以下の各形式では，関数群を規定しています。関数群には，主要な関数 (1 つ以上の double complex 型のパラメータと， double complex または double 型の返却値を持つ) が 1 つあります。また，同じ名前で f および l の接尾語が付いた関数は，パラメータおよび返却値が float 型および long double 型の，対応する関数です。

マクロ

complex

_Complexに展開されます。

_Complex_I

虚数単位の値を持つ，const float _Complex型の定数式に展開されます。虚数単位とは， $i^2 = -1$ になる数 i です。

imaginary

_Imaginary_I

処理系が虚数型をサポートしている場合だけ定義されます。これらのマクロは，定義されている場合，_Imaginaryと，虚数単位の値を持つconst float _Imaginary型の定数式に展開されます。

I

`_Imaginary_I`または`_Complex_I`に展開されます。`_Imaginary_I`が定義されていない場合、`I`は`_Complex_I`に展開されます。

三角関数

`cacos`関数

```
#include <complex.h>
double complex cacos(double complex z);
float complex cacosf(float complex z);
long double complex cacosl(long double complex z);
```

`cacos`関数は、複素数 z の逆余弦を、分岐線法で実数軸 $[-1, +1]$ の外側を除いて算出します。

`cacos`関数は、複素数 z の逆余弦をラジアン単位の値で返します。この値は、虚数軸に対しては数学的に無限の範囲にあり、実数軸では $[0, \pi]$ の範囲にあります。

`casin`関数

```
#include <complex.h>
double complex casin(double complex z);
float complex casinf(float complex z);
long double complex casinl(long double complex z);
```

`casin`関数は、複素数 z の逆正弦を、分岐線法で実数軸 $[-1, +1]$ の外側を除いて算出します。

`casin`関数は、複素数 z の逆正弦をラジアン単位の値で返します。この値は、虚数軸に対しては数学的に無限の範囲にあり、実数軸では $[-\pi/2, +\pi/2]$ の範囲にあります。

`catan`関数

```
#include <complex.h>
double complex catan(double complex z);
float complex catanf(float complex z);
long double complex catanl(long double complex z);
```

`catan`関数は、複素数 z の逆正接を、分岐線法で虚数軸 $[-i, +i]$ の外側を除いて算

出します。

catan関数は、複素数 z の逆正接をラジアン単位で返します。この値は、虚数軸に対しては数学的に無限の範囲にあり、実数軸では $[-\pi/2, +\pi/2]$ の範囲にあります。

ccos関数

```
#include <complex.h>
double complex ccos(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

ccos関数は、複素数 z の余弦を返します。

csin関数

```
#include <complex.h>
double complex csin(double complex z);
float complex csinf(float complex z);
long double complex csinl(long double complex z);
```

csin関数は、複素数 z の正弦を返します。

ctan関数

```
#include <complex.h>
double complex ctan(double complex z);
float complex ctanf(float complex z);
long double complex ctanl(long double complex z);
```

ctan関数は、複素数 z の正接を返します。

双曲線関数

cacosh関数

```
#include <complex.h>
double complex cacosh(double complex z);
float complex cacoshf(float complex z);
long double complex cacoshl(long double complex z);
```

cacosh関数は、複素数 z の双曲線逆余弦を、分岐線法で実数軸 1 未満の値を除いて算出します。

ANSI C 標準ライブラリ

9.2 複素数算術 (<complex.h>)

`cacosh`関数は、複素数 z の双曲線逆余弦をラジアン単位の値で返します。この値は、実数軸の非負側の範囲にあり、虚数軸では $[-i\pi, +i\pi]$ の範囲にあります。

`casinh`関数

```
#include <complex.h>
double complex casinh(double complex z);
float complex casinhf(float complex z);
long double complex casinhl(long double complex z);
```

`casinh`関数は、複素数 z の双曲線逆正弦を、分岐線法で虚数軸 $[-i, +i]$ の外側を除いて算出します。

`casinh`関数は、複素数の双曲線逆正弦の値を返します。この値は、実数軸に対しては数学的に無限の範囲にあり、虚数軸では $[-i\pi/2, +i\pi/2]$ の範囲にあります。

`catanh`関数

```
#include <complex.h>
double complex catanh(double complex z);
float complex catanhf(float complex z);
long double complex catanhl(long double complex z);
```

`catanh`関数は、複素数 z の双曲線逆正接を、分岐線法で実数軸 $[-1, +1]$ の外側を除いて算出します。

`catanh`関数は、複素数の双曲線逆正接の値を返します。この値は、実数軸に対しては数学的に無限の範囲にあり、虚数軸では $[-i\pi/2, +i\pi/2]$ の範囲にあります。

`ccosh`関数

```
#include <complex.h>
double complex ccosh(double complex z);
float complex ccoshf(float complex z);
long double complex ccoshl(long double complex z);
```

`ccosh`関数は、複素数 z の双曲線余弦を返します。

csinh関数

```
#include <complex.h>
double complex csinh(double complex z);
float complex csinhf(float complex z);
long double complex csinhl(long double complex z);
```

csinh関数は、複素数 z の双曲線正弦を返します。

ctanh関数

```
#include <complex.h>
double complex ctanh(double complex z);
float complex ctanhf(float complex z);
long double complex ctanhl(long double complex z);
```

ctanh関数は、複素数 z の双曲線正接を返します。

指数関数および自然対数関数

cexp関数

```
#include <complex.h>
double complex cexp(double complex z);
float complex cexpf(float complex z);
long double complex cexpl(long double complex z);
```

cexp関数は、複素数 z の基数 e の指数を返します。

clog関数

```
#include <complex.h>
double complex clog(double complex z);
float complex clogf(float complex z);
long double complex clogl(long double complex z);
```

clog関数は、複素数 z の自然対数 (基数 e) を、分岐線法で負の実数軸に沿って除外して算出します。

clog関数は、複素数の自然対数値を返します。この値は、実数軸に対しては数学的に無限の範囲にあり、虚数軸では $[-i\pi, +i\pi]$ の範囲にあります。

累乗関数および絶対値関数

cabs関数

```
#include <complex.h>
double cabs(double complex z);
float cabsf(float complex z);
long double cabsl(long double complex z);
```

cabs関数は、複素数 z の絶対値 (ノルム, モジューロ, またはマグニチュードともいう) を返します。

cpow関数

```
#include <complex.h>
double complex cpow(double complex x, double complex y);
float complex cpowf(float complex x, float complex y);
long double complex cpowl(long double complex x,
                           long double complex y);
```

cpow関数は、複素数の累乗関数 xy を、分岐線法で負の実数軸に沿って1番目のパラメータを除外して算出します。

cpow関数は、複素数の累乗関数値を返します。

csqrt関数

```
#include <complex.h>
double complex csqrt(double complex z);
float complex csqrtf(float complex z);
long double complex csqrtl(long double complex z);
```

csqrt関数は、複素数 z の平方根を、分岐線法で負の実数軸に沿って除外して算出します。

csqrt関数は複素数の平方根値を返却します。この値は、複素平面の右半分 (虚数軸を含む) の範囲にあります。

操作関数

carg関数

```
#include <complex.h>
double carg(double complex z);
float cargf(float complex z);
long double cargl(long double complex z);
```

carg関数は、 z の偏角 (位相角ともいう) を、分岐線法で負の実数軸に沿って除外して算出します。

carg関数は、偏角の値を返します。この値は、 $[-\pi, +\pi]$ の範囲にあります。

cimag関数

```
#include <complex.h>
double cimag(double complex z);
float cimagf(float complex z);
long double cimagl(long double complex z);
```

cimag関数は、 z の虚数部を算出し、実数として返します。

conj関数

```
#include <complex.h>
double complex conj(double complex z);
float complex conjf(float complex z);
long double complex conjl(long double complex z);
```

conj関数は、複素数 z の共役複素数を、虚数部の符号を逆にして算出します。

conj関数は、複素数の共役複素数値を返します。

cproj関数

```
#include <complex.h>
double complex cproj(double complex z);
float complex cprojf(float complex z);
long double complex cprojl(long double complex z);
```

cproj関数は、 z の、リーマン球面への射影を算出します。すべての複素数の無限大 (無限大部が 1 つと NaN 部が 1 つのものも含む) が実数軸上の正の無限大

に射影するのを除き、 z は z に射影します。 z に無限大部がある場合、 $cproj(z)$ は次の式と同じになります。

$$\text{INFINITY} + I * \text{copysign}(0.0, \text{cimag}(z))$$

$cproj$ 関数は、リーマン球面への射影の値を返します。

$creal$ 関数

```
#include <complex.h>
double creal(double complex z);
float crealf(float complex z);
long double creall(long double complex z);
```

$creal$ 関数は、 z の実数部を返します。

複素数型の変数 z について、 $z == \text{creal}(z) + \text{cimag}(z)*I$ が成り立ちます。

9.3 文字処理 (<ctype.h>)

<ctype.h>ヘッダ・ファイルは、文字を検査するための複数の関数を宣言します。各関数とも実引数は `int` であり、値は `EOF` または `unsigned char` として表現可能なものでなければなりません。返却値は整数です。

関数

```
int isalnum(int c);
```

この関数へ渡された文字が ASCII の英数字である場合、0 以外の整数を返す。それ以外の場合には、`isalnum` は 0 を返す。

```
int isalpha(int c);
```

この関数へ渡された文字が ASCII の英字である場合、0 以外の整数を返す。それ以外の場合には、`isalpha` は 0 を返す。

```
int iscntrl(int c);
```

この関数へ渡された文字が ASCII の DEL 文字 (8 進の 177, 16 進の 0x7F) または非印字の ASCII 文字 (8 進の 40, 16 進の 0x20 よりも小さいコード) である場合、0 以外の整数を返す。それ以外の場合には、`iscntrl` は 0 を返す。

```
int isdigit(int c);
```

この関数へ渡された文字が 10 進数字文字 (0 から 9) である場合、0 以外の整数を返す。それ以外の場合には、isdigit は 0 を返す。

```
int isgraph(int c);
```

この関数へ渡された文字が ASCII の図形文字 (空白文字以外の印字文字) である場合、0 以外の整数を返す。それ以外の場合には、isgraph は 0 を返す。

```
int islower(int c);
```

この関数へ渡された文字が ASCII の英小文字である場合、0 以外の整数を返す。それ以外の場合には、islower は 0 を返す。

```
int isprint(int c);
```

この関数へ渡された文字が空白文字を含む ASCII の印字文字である場合、0 以外の整数を返す。それ以外の場合には、isprint は 0 を返す。

```
int ispunct(int c);
```

この関数へ渡された文字が ASCII の区切り文字 (英数字以外の印字文字で 8 進の 40、16 進の 0x20 よりも大きいコード) である場合、0 以外の整数を返す。それ以外の場合には、ispunct は 0 を返す。

```
int isspace(int c);
```

この関数へ渡された文字が空白である場合、0 以外の整数を返す。それ以外の場合には、isspace は 0 を返す。標準空白文字は次のとおり。

- 空白 (' ')
- フォーム・フィード ('\f')
- 改行 ('\n')
- キャリッジ・リターン ('\r')
- 水平タブ ('\t')
- 垂直タブ ('\v')

```
int isupper(int c);
```

この関数へ渡された文字が ASCII の英大文字である場合，0 以外の整数を返す。それ以外の場合には，isupper は 0 を返す。

```
int isxdigit(int c);
```

この関数へ渡された文字が 16 進数字 (0 ~ 9, A ~ F, または a ~ f) である場合，0 以外の整数を返す。それ以外の場合には，isxdigit は 0 を返す。

```
int tolower(int c);
```

大文字を小文字に変換する。c は大文字以外の場合には変更されない。

```
int toupper(int c);
```

小文字を大文字に変換する。c は小文字以外の場合には変更されない。

9.4 エラー・コード (<errno.h>)

<errno.h>ヘッダ・ファイルは，エラー報告に使用する複数のマクロを定義します。

マクロ

EDOM

ERANGE

errno に格納できるエラー・コード。これは，0 以外の固有の値を持つ汎整数定数式に展開される。

変数またはマクロ

errno

使用するオペレーティング・システムに応じて，int型で可変左辺値に展開される外部変数またはマクロ。

errno変数は，ライブラリ・ルーチンからの処理系定義のエラー・コードを保持するために使用する。エラー・コードはすべて正の整数。errno の値はプログラム開始時には 0 だが，ライブラリ関数によって 0 にセットされることはな

い。したがって、`errno` はライブラリ関数を呼び出す前に 0 にセットし、その後で検査を行う必要がある。

9.5 ANSI C の限界 (<limits.h>および<float.h>)

<limits.h> および <float.h> ヘッダ・ファイルは、それぞれの処理系固有の限界および仮引数に展開される複数のマクロを定義します。これらのマクロのほとんどは、ハードウェアの整数および浮動小数点の特性を記述します。詳細については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

9.6 ローカル化 (<locale.h>)

<locale.h> ヘッダ・ファイルは 2 つの関数と 1 つの型を宣言し、複数のマクロを定義します。

型

`struct lconv`

数値の書式化に関連するメンバを含んでいる構造体。この構造体は次のメンバを任意の順序で含む。それぞれの値をコメント中に示す。

ANSI C 標準ライブラリ

9.6 ローカル化 (<locale.h>)

```
char *decimal_point;      /* "." */
char *thousands_sep;     /* "" */
char *grouping;           /* "" */
char *int_curr_symbol;    /* "" */
char *currency_symbol;    /* "" */
char *mon_decimal_point;  /* "" */
char *mon_thousands_sep; /* "" */
char *mon_grouping;       /* "" */
char *positive_sign;      /* "" */
char *negative_sign;      /* "" */
char int_frac_digits;     /* CHAR_MAX */
char frac_digits;         /* CHAR_MAX */
char p_cs_precedes;       /* CHAR_MAX */
char p_sep_by_space;      /* CHAR_MAX */
char n_cs_precedes;       /* CHAR_MAX */
char n_sep_by_space;      /* CHAR_MAX */
char p_sign_posn;         /* CHAR_MAX */
char n_sign_posn;         /* CHAR_MAX */
```

これらのメンバについては、この節の `localeconv` 関数の項で説明します。

マクロ

```
NULL
LC_ALL
LC_COLLATE
LC_CTYPE
LC_MONETARY
LC_NUMERIC
LC_TIME
```

それぞれ異なる値で汎整数定数式に展開し、`setlocale` 関数の最初の実引数として使用できる。

関数

```
char *setlocale(int category, const char *locale);
```

category および *locale* 実引数によって指定された、プログラムのロケール (地域) の適切な部分を選択する。この関数は、プログラムの現在のロケール全体またはその一部を変更したり照会する場合に使用できる。

category 実引数には次の値を指定できる。

- LC_ALL — プログラムのロケール全体に影響を及ぼす。
- LC_COLLATE — `strcoll` および `strxfrm` 関数の動作に影響を及ぼす。
- LC_CTYPE — 文字処理関数および多バイト関数の動作に影響を及ぼす。
- LC_MONETARY — `localeconv` 関数によって返される通貨の書式化情報に影響を及ぼす。
- LC_NUMERIC — 書式付き入出力関数および文字列変換関数の小数点文字、および `localeconv` 関数によって返される非通貨数量の書式化情報に影響を及ぼす。
- LC_TIME — `strftime` 関数の動作に影響を及ぼす。

locale 実引数には次の値を指定できる。

- "C" — C 言語の翻訳の最低限の環境を指定する。
- "" — *category* に対応する環境変数の使用を指定する。この環境変数をセットしない場合には、LANG 環境変数を使用する。LANG もセットしなければ、エラーが返される。

プログラムの開始処理時に次に示す関数が実行される。

```
setlocale(LC_ALL, "C");
```

`setlocale` 関数は次のいずれか 1 つを返す。

- 文字列へのポインタを *locale* として指定し、この選択が受け入れられた場合、`setlocale` は指定された *category* に関連する文字列へのポインタを新しいロケールとして返す。この選択が受け入れられなかった場合には、`setlocale` は空ポインタを返し、プログラムのロケールは変更されない。
- 空ポインタを *locale* として指定した場合、`setlocale` は *category* に関連する文字列へのポインタをプログラムの現在のロケールとして返す。プログラムのロケールは変更されない。

返される文字列へのポインタはいずれの場合も、後続の呼出しでこの文字列値とその関連カテゴリを使用することによって、プログラムのロケールのその部分がリストアされる。この文字列をプログラムによって変更することはできないが、後続の `setlocale` の呼出しによって重ね書きできる。

```
struct lconv *localeconv(void);
```

struct lconv型のオブジェクトの構成要素を、現在のロケールの規則に従って数値を書式化するのに適切な値でセットする。

char*型の構造体メンバは文字列へのポインタであり、decimal_point 以外のいずれのメンバも "" を示すことができる。"" は値の長さが0であるか、現在のロケールで使用不可能であることを示す。char型の構造体メンバは非負数であり、そのいずれも CHAR_MAX となって、値が現在のロケールで使用不可能であることを示すことができる。構造体メンバには次のものがある。

```
char *decimal_point
```

非通貨数量の書式化に使用する小数点文字。

```
char *thousands_sep
```

書式化された非通貨数量の小数点の前の一連の数字を区切る文字。

```
char *grouping
```

各要素が書式化された非通貨数量の一連の数字のサイズを示す文字列。

```
char *int_curr_symbol
```

現在のロケールに適用される国際通貨記号。先頭の3文字は、ISO 4217 Codes for the Representation of Currency and Funds の指定に準拠した英字の国際通貨記号を含む。4番目の文字(ヌル文字の直前)は、国際通貨記号と通貨数量を区切る文字。

```
char *currency_symbol
```

現在のロケールに適用する現地通貨記号。

```
char *mon_decimal_point
```

通貨数量の書式化に使用する小数点文字。

`char *mon_thousands_sep`

書式化された通貨数量の小数点の前の一連の数字を区切る文字。

`char *mon_grouping`

各要素が書式化された通貨数量の一連の数字のサイズを示す文字列。

`char *positive_sign`

非負数の書式化された通貨数量を示す文字列。

`char *negative_sign`

負数の書式化された通貨数量を示す文字列。

`char int_frac_digits`

書式化された国際通貨数量で表示される少数桁の数。

`char frac_digits`

書式化された通貨数量で表示される少数桁の数。

`char p_cs_precedes`

`currency_symbol` が非負数の書式化された通貨数量の値に先行する場合には、1 にセットされる。`currency_symbol` がこの値の後に続く場合には、0 にセットされる。

`char p_sep_by_space`

`currency_symbol` が非負数の書式化された通貨数量の値から空白で区切られている場合には、1 にセットされる。空白がない場合には0 にセットされる。

char n_cs_precedes

currency_symbol が負数の書式化された通貨数量の値に先行する場合には、1 にセットされる。currency_symbol がこの値の後に続く場合には 0 にセットされる。

char n_sep_by_space

currency_symbol が負数の書式化された通貨数量の値から空白で区切られている場合には、1 にセットされる。空白がない場合には 0 にセットされる。

char p_sign_posn

非負数の書式化された通貨数量の positive_sign の位置決めを示す値にセットされる。

char n_sign_posn

負数の書式化された通貨数量の negative_sign の位置決めを示す値にセットされる。

grouping および mon_grouping の要素は、次のように解釈される。

- CHAR_MAX — これ以上のグループ化は行われない。
- 0 — 残りの数字に前の要素を繰り返し使用する。
- その他 — 整数値は現在のグループを構成する数字の数である。現在のグループの前に、次のグループのサイズを決めるために次の要素を調べる。

p_sign_posn および n_sign_posn の値は、次のように解釈される。

- 0 — 数量と currency_symbol を括弧で囲む。
- 1 — 符号文字列が数量と currency_symbol に先行する。
- 2 — 符号文字列が数量と currency_symbol の後に続く。
- 3 — 符号文字列が currency_symbol の直前にくる。
- 4 — 符号文字列が currency_symbol の直後に続く。

localeconv関数は構造体の空ではない部分へのポインタを返す。構造体をプログラムによって変更することはできないが、localeconv または setlocale へのその後の呼出しで LC_ALL, LC_MONETARY, または LC_NUMERIC カテゴリを指定することによって、重ね書きできる。

9.7 算術 (<math.h>)

<math.h>ヘッダ・ファイルは、型、マクロと複数の算術関数を定義します。関数は double実引数を取り、倍精度値を返します。

このヘッダ中の各関数の動作は、それぞれの入力実引数のすべての表現可能な値に対して定義されます。各関数は単一演算と同様に実行され、外部的に認識できる例外は生じません。

すべての関数について、算術関数が定義されている定義域の外側に入力実引数がある場合、定義域エラーが起こります。定義域エラーが起こるかどうかは各関数の説明に記載しています。定義域エラーが起こると関数は処理系定義の値を返し、EDOMマクロの値が errno に格納されます。

すべての関数について、関数の結果を double値として表現できない場合、範囲エラーが起こります。結果がオーバーフローした場合 (結果が大きすぎて、指定した型のオブジェクト内で表現できない場合)、関数は関数の正しい値と同じ符号で (tan関数の場合を除く)、HUGE_VALマクロの値を返します。ERANGEマクロの値が errno に格納されます。結果がアンダフローした場合 (結果が小さすぎて、指定した型のオブジェクト内で表現できない場合)、関数は 0 を返します。ERANGEマクロの値が errno に格納されるかどうかは、それぞれの処理系定義によって異なります。

マクロ

HUGE_VAL

正の double式に展開する。

INFINITY

可能な場合は、正または符号なしの無限大を示すfloat型の定数式に展開する。それ以外では、変換時にオーバーフローになるfloat型の正の定数に展開する。

NAN

Quiet NaN を示すfloat型の定数式に展開する。

三角関数

```
double acos(double x);
```

x の逆余弦の値を $[0, \pi]$ の範囲でラジアン単位で返す。実引数が $[-1, +1]$ の間
にない場合、定義域エラーが起こる。

```
double asin(double x);
```

x の逆正弦の値を $[-\pi/2, +\pi/2]$ の範囲でラジアン単位で返す。実引数が
 $[-1, +1]$ の間にない場合、定義域エラーが起こる。

```
double atan(double x);
```

x の逆正接の値を $[-\pi/2, +\pi/2]$ の範囲でラジアン単位で返す。

```
double atan2(double y, double x);
```

y/x の逆正接の値をラジアン単位で返し、両方の実引数の符号を使用して返却値
の象限を決める。返される値の範囲は $[-\pi, +\pi]$ である。両方の実引数が0の
場合、定義域エラーが起こることがある。

```
double cos(double x);
```

x の余弦の値をラジアン単位で返す。

```
double sin(double x);
```

x の正弦の値をラジアン単位で返す。

```
double tan(double x);
```

x の正接の値をラジアン単位で返す。

双曲線関数

`double cosh(double x);`

x の双曲線余弦の値を返す。 x が大きすぎる場合、範囲エラーが起こる。

`double sinh(double x);`

x の双曲線正弦の値を返す。 x が大きすぎる場合、範囲エラーが起こる。

`double tanh(double x);`

x の双曲線正接の値を返す。

指数関数と対数関数

`double exp(double x);`

x の指数関数の値を返す。 x が大きすぎる場合、範囲エラーが起こる。

`double frexp(double value, int *eptr);`

value 浮動小数点数を 0 または $[1/2, 1]$ の間にある正規化した小数部、および 2 の汎整数累乗に分割する。この関数は小数部を返し、2 の汎整数累乗は *eptr* が示す `int` オブジェクトに格納される。*value* が 0 の場合、結果は両方の部分とも 0 になる。

`double ldexp(double x, int exp);`

浮動小数点数を 2 の汎整数累乗で乗算し、 $x \times 2^{\text{exp}}$ を返す。範囲エラーが起こることがある。

`double log(double x);`

x の自然対数を返す。実引数が負の場合、定義域エラーが起こる。実引数が 0 の場合には、範囲エラーが起こることがある。

`double log10(double x);`

x の底を 10 とする対数を返す。 x が負の場合、定義域エラーが起こる。 x が 0 の場合には、範囲エラーが起こることがある。

```
double modf(double value, double *iptr);
```

value 実引数を整数部と小数部に分割する。いずれの部分も実引数と同じ符号を持つ。modf関数は符号付き小数部を返し、iptr が示すオブジェクトに double として整数部を格納する。

累乗関数

```
double pow(double x, double y);
```

x^y を返す。 x が負で、 y が汎整数値以外の場合、定義域エラーが起こる。 x が 0 で y が 0 以下のときに結果を表現できない場合、定義域エラーが起こる。範囲エラーが起こることがある。

```
double sqrt(double x);
```

x の非負数の平方根を返す。 x が負の場合、定義域エラーが起こる。

整数の近似値、絶対値、および剰余関数

```
double ceil(double x);
```

x 以上の最小の汎整数値を返す。

```
double fabs(double x);
```

x 浮動小数点数の絶対値を返す。

```
double floor(double x);
```

x 以下の最大の汎整数値を返す。

```
double fmod(double x, double y);
```

x/y の浮動小数点剰余を計算する。fmod関数は、 i を整数とすると $x - i * y$ を返す。 y が 0 以外の場合、結果は x と同じ符号になり、大きさは y よりも小さくなる。 y が 0 の場合、関数は 0 を返す。

9.8 非ローカル飛越し (<setjmp.h>)

<setjmp.h>ヘッダ・ファイルには、通常の間数呼出しと戻りの処理を行わずに、ネストされた間数呼出しから途中で戻ることを可能にする方法を提供する宣言が含まれています。

マクロ

```
int setjmp(jmp_buf env)
```

ローカルの `jmp_buf` バッファを設定し、飛越しのために初期化する (飛越し自体は `longjmp` で実行される)。このマクロは、プログラムの呼出し環境を `env` 実引数によって指定された環境バッファに保存し、後に `longjmp` 関数がこれを使用できるようにする。 `setjmp` 関数の直接の呼出しの場合、 `setjmp` は 0 を返す。
`longjmp` への呼出しからの戻りの場合、 `setjmp` は 0 以外の値を返す。

型

```
jmp_buf
```

呼出し環境をリストアするために必要な情報を保持するのに適切な配列型。

関数

```
void longjmp(jmp_buf env, int value);
```

プログラムの同じ呼出し中に `setjmp` 関数の呼出しによって保存された、 `env` 環境バッファのコンテキストをリストアする。 `longjmp` 関数はネストされたシグナル・ハンドラから呼び出された場合には動作せず、その動作結果は定義されていない。

`value` によって指定された値は、 `longjmp` から `setjmp` へ渡される。 `longjmp` の完了後、それに対応する `setjmp` の呼出しによって `value` が返された場合と同様に、プログラムは続行する。 `value` が 0 として `setjmp` に渡された場合には、1 に変換される。

9.9 シグナル処理 (<signal.h>)

<signal.h>ヘッダ・ファイルは 1 つの型と 2 つの間数を宣言し、プログラム実行中に報告される可能性がある例外条件を処理するための複数のマクロを定義します。

型

sig_atomic_t

非同期割込みがあった場合でも，アトミック要素としてアクセスできるオブジェクトの汎整数型。

マクロ

SIG_DFL

SIG_ERR

SIG_IGN

signal関数の第2実引数，およびこの関数の返却値と互換性がある型を持つ値の定数式にそれぞれ展開する。この値は，宣言可能な関数のアドレスとは等しくない。

関数

void (*signal(int sig, void (*handler) (int))) (int);

後続のシグナルの処理方法を定める。シグナルは次のように処理される。

1. handler の値が SIG_DFL の場合，そのシグナルの省略時の処理が起こる。
2. handler の値が SIG_IGN の場合，シグナルは無視される。
3. 前記以外の場合にこのシグナルが発生すると，handler が示す関数がシグナルの型の実引数で呼び出される。このような関数をシグナル・ハンドラと呼ぶ。有効なシグナルは次のとおり。
 - SIGABRT — 異常終了。たとえば，abort関数。
 - SIGFPE — 算術エラー。たとえば，0 除算またはオーバフロー。
 - SIGILL — 無効な関数イメージ。たとえば，無効な命令。
 - SIGINT — 対話型介入。たとえば，割込み。
 - SIGSEGV — 記憶域への無効なアクセス。たとえば，メモリ限界の外側へのアクセス。
 - SIGTERM — プログラムへ送られる終了要求。

その他のシグナルは，各オペレーティング・システムに依存する。

要求が受け入れられた場合、signal関数は指定した sigシグナルに対する signal への最新の呼出しに対して handler の値を返す。要求が受け入れられなかった場合には、SIG_ERR の値が返され、処理系定義の正の値が errno に格納されます。

```
int raise(int sig);
```

sigシグナルを実行プログラムに送る。raise関数は成功した場合には 0 を返し、成功しなかった場合には 0 以外の値を返す。

9.10 可変個実引数 (<stdarg.h>)

<stdarg.h>ヘッダ・ファイルは 1 つの型を宣言し、3 つのマクロを定義して、可変個の数と型を持つ関数の実引数並びの処理を行います。

型

va_list

va_start, va_arg, および va_endマクロが必要とする情報を保持するのに適切な型。

可変個の実引数にアクセスするには、呼び出された関数は va_list型のオブジェクトを宣言しなければならない。次の例では、apがオブジェクト。

```
va_list ap;
```

apオブジェクトは、実引数として別の関数に渡すことができる。その関数が ap実引数で va_argマクロを呼び出した場合、呼出し関数の中の ap の値は不定になり、ap への次の参照を行う前に va_endマクロへ渡される。

マクロ

```
void va_start(va_list ap, parmN);
```

ap を初期化し、それ以後に va_arg および va_end が使用できるようにする。未指名の実引数にアクセスする前には、va_startマクロを呼び出さなければならない。

parmN 仮引数は、関数定義の可変個仮引数並び中の右端の仮引数の識別子。
parmN を register 記憶域クラス、関数型または配列型、あるいは省略時の実引数拡張の適用後に生じる型と互換性を持たない型で宣言した場合の動作結果は定義されていない。va_start マクロは値を返さない。

```
type va_arg(va_list ap, type);
```

呼出し中の次の実引数の型と値を持つ式に展開する。*ap* 仮引数は va_start によって初期化された va_list ap と同じ。va_arg を呼び出すたびに *ap* が変更され、連続する実引数の値が順に返される。*type* 仮引数は指定された型名であり、*type* の後にアスタリスク (*) を付けることによって、指定した型のオブジェクトへのポインタの型を取得することができる。次の実引数がない場合、または *type* が (省略時の実引数拡張によって拡張された) 次の実引数の型と互換性を持たない場合の動作結果は定義されていない。

va_start の呼出しの後に初めて va_arg を呼び出すと、*parmN* によって指定された後の実引数の値を返す。連続して呼び出すと、残りの実引数の値を順に返す。

```
void va_end(va_list ap);
```

va_list ap オブジェクトを初期化した va_start の展開によって可変個実引数並びが参照された関数から通常の返却値を返す。va_end マクロは *ap* を変更して、(va_start の呼出しなしには) 使用できないようにすることができる。対応する va_start が呼び出されなかった場合、または戻る前に va_end が呼び出されなかった場合の動作結果は定義されていない。va_end マクロは値を返さない。

9.11 ブール型とブール値 (<stdbool.h>)

<stdbool.h> ヘッド・ファイルは、マクロを 4 つ定義しています。

マクロ

`bool`

`_Bool`に展開されます。

`true`

`false`

`__bool_true_false_are_defined`

`#if`前処理命令内で使用するのに適しています。

`true`は、整数定数 1 に展開されます。

`false`は、整数定数 0 に展開されます。

`__bool_true_false_are_defined`は、整数定数 1 に展開されます。

9.12 共通定義 (<stddef.h>)

<stddef.h>ヘッダ・ファイルは複数の型とマクロを定義し、そのいくつかは他のヘッダ・ファイルにも定義されています。

型

`ptrdiff_t`

2 つのポインタの差を表す符号付き汎整数型。

`size_t`

`sizeof`演算子の結果を表す符号なし汎整数型。

`wchar_t`

汎整数型。値の範囲は、サポートされるロケールに指定された最大の拡張文字集合のすべてのメンバを個別のコードで表すことができる。

マクロ

NULL

処理系定義の空ポインタ定数に展開する。

`offsetof(type, member-designator)`

値を (type で指定した) 構造体の先頭から (member-designator で指定した) 構造体メンバへのバイト単位のオフセットとして `size_t` 型の汎整数定数式に展開する。member-designator は、次に示すような式 `&(t.member-designator)` をアドレス定数として評価される。

`static type t;`

指定したメンバがビット・フィールドの場合の動作結果は定義されていない。

9.13 標準入出力 (<stdio.h>)

<stdio.h> ヘッダ・ファイルは、テキストの入出力を行うための 3 つの型、複数のマクロ、および多数の関数を宣言します。テキスト・ストリームは一連の行からなり、各行は改行文字で終わります。

型

`size_t`

`sizeof` 演算子の結果の符号なし汎整数型。

`FILE`

データ・ストリームの制御に必要なすべての情報を記録することができるオブジェクト型。これは、ファイル位置指示子、その関連バッファ (存在する場合) へのポインタ、読み込み/書き込みエラーが起きたかどうかを記録するエラー指示子、およびファイルの終わりに達したかどうかを記録するファイル終了指示子を含む。

`fpos_t`

ファイル内の各位置を個別に指定するために必要なすべての情報を記録できるオブジェクト。

マクロ

NULL

処理系定義の空ポインタ定数に展開する。

_IOFBF

_IOLBF

_IONBF

setvbuf関数の第3実引数として使用するために、適切なそれぞれの値の汎整数定数式に展開する。

BUFFSIZ

汎整数定数式に展開する。この式は、setbuf関数が使用するバッファのサイズ。

EOF

負の汎整数定数式に展開する。これは、多数の関数がファイルの終わりを示すために返す。

FOPEN_MAX

汎整数定数式に展開する。これは、*Compaq C* コンパイラが各システムに対して、同時にオープンできることを保証するファイルの最大数。

FILENAME_MAX

汎整数定数式に展開する。これは、*Compaq C* コンパイラが各システムでオープンできる最長のファイル名文字列を char の配列で保持するのに必要なサイズ。

L_tmpnam

汎整数定数式に展開する。これは、tmpnam関数によって作成された一時的ファイル名の文字列を char の配列で保持するのに必要なサイズ。

ANSI C 標準ライブラリ
9.13 標準入出力 (<stdio.h>)

SEEK_CUR
SEEK_END
SEEK_SET

fseek関数の第3実引数として使用するために、適切なそれぞれの値の汎整数定数式に展開する。

TMP_MAX

tmpnam関数が生成できる固有のファイル名の最大数である汎整数定数式に展開する。

stderr
stdin
stdout

標準エラー，入力および出力ストリームに関連する FILEオブジェクトをそれぞれ示す FILE へのポインタ型の式。

ファイル操作関数

```
int remove(const char *filename);
```

filename が示す名前を持つファイルを，この名前でアクセスできないようにする。これ以降この名前でこのファイルをオープンしようとするとう失敗する。
remove関数は操作が成功した場合には0を返し，失敗した場合には0以外を返す。ファイルがオープンされた場合，この関数の動作は各処理系定義により異なる。

```
int rename(const char *old, const char *new);
```

ファイルの名前を，*old* が示す名前から *new* が示す名前に変更する。このファイルは，古いファイル名ではアクセス不可能になる。rename関数は操作が成功した場合には0を返し，失敗した場合には0以外を返す。ただし，失敗した場合でそのファイルが存在している場合には，元の名前のままで識別される。
rename を呼び出す前に新しいファイルが存在している場合には，この関数の動作は各処理系定義により異なる。

```
FILE *tmpfile(void);
```

ファイルのクローズ時やプログラム実行の終了時に自動的に削除される一時的バイナリ・ファイルを作成する。実行が異常終了した場合には、オープンしているその一時的ファイルを削除するかどうかは各処理系によって異なる。更新のため、ファイルのオープンは wb+モードで行われる (表 9-1 を参照のこと)。tmpfile関数は作成したファイルのストリームへのポインタを返す。ファイルを作成できない場合には、tmpfile は空ポインタを返す。

```
FILE *tmpnam(void);
```

既存のファイルの名前とは異なる有効なファイル名を作成する。tmpnam の呼出しを最高で TMP_MAX回まで行つたびに、異なる名前が作成される。tmpnam の呼出しを TMP_MAX回よりも多く行った場合の動作は、各処理系定義により異なる。

実引数が空ポインタの場合には、tmpnam関数はその結果を内部の静的オブジェクトに残し、このオブジェクトへのポインタを返す。これ以降の tmpnam の呼出しで、この同じオブジェクトを変更することができる。実引数が空ポインタではない場合には、少なくとも L_tmpnam chars の配列へのポインタとみなされる。tmpnam関数はその結果をこの配列へ書き込み、実引数をその値として返す。

ファイル・アクセス関数

```
int fclose(FILE *stream);
```

stream が示すストリームをフラッシュし、関連ファイルをクローズする。そのストリームの未書込みのバッファ内に格納されているデータは、ホスト環境へ転送され、ファイルへ書き込まれる。未読込みのバッファ内に格納されているデータは破棄される。ストリームとファイルとの関連が解除される。関連バッファが自動的に割り当てられていた場合、この割り当ては解除される。fclose関数はストリームのクローズが成功した場合には 0 を返し、エラーが検出された場合には EOF を返す。

```
int fflush(FILE *stream);
```

stream が出力ストリームまたは更新ストリーム (最新の操作が入力ではない) を示す場合、fflush関数はまだ書き込んでいないデータをホスト環境に転送し、

ファイルに書き込む。それ以外の場合の動作結果は定義されていない。*stream* が空ポインタの場合には、*fflush* は出力ストリームまたは更新ストリーム (最新の操作が入力ではない) のすべてをフラッシュする。*fflush*関数は操作が成功した場合は 0 を返し、書き込みエラーが起きた場合は EOF を返す。

```
FILE *fopen(const char *filename, const char *mode);
```

filename が示すファイルをオープンし、そのファイルにストリームを関連付ける。*mode*実引数は、表 9-1 に示した文字列の 1 つで始まる文字列を示す。これらの文字列には追加の文字が続くことがある。

表 9-1 ファイル・モード

モード	説明
r	読み込みのためにテキスト・ファイルをオープンする。
w	長さ 0 に切り捨てるか、または書き込みのためにテキスト・ファイルを作成する。
a	追加。ファイルの終わりに書き込むためにテキスト・ファイルをオープンまたは作成する。
rb	読み込みのためにバイナリ・ファイルをオープンする。
wb	長さ 0 に切り捨てるか、または書き込みのためにバイナリ・ファイルを作成する。
ab	追加。ファイルの終わりに書き込むためにバイナリ・ファイルをオープンまたは作成する。
r+	更新 (読み込みおよび書き込み) のためにテキスト・ファイルをオープンする。
w+	長さ 0 に切り捨てるか、または更新のためにテキスト・ファイルを作成する。
a+	追加。更新とファイルの終わりに書き込むためにテキスト・ファイルをオープンまたは作成する。
r+b または rb+	更新 (読み込みおよび書き込み) のためにバイナリ・ファイルをオープンする。
w+b または wb+	長さ 0 に切り捨てるか、または更新のためにバイナリ・ファイルを作成する。

(次ページに続く)

表 9-1 (続き) ファイル・モード

モード	説明
a+b または ab+	追加。更新とファイルの終わりに書き込むためにバイナリ・ファイルをオープンまたは作成する。

fopen関数はストリームを制御するオブジェクトへのポインタを返す。オープン操作が失敗した場合、fopen は空ポインタを返す。

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

filename が示すファイルをオープンして、*stream* が示すストリームをこのファイルに関連付ける。*mode*実引数は、fopen関数と同様に使用される。freopen関数は最初に、指定したストリームに関連するファイルをクローズしようとする。ファイルのクローズが成功しなかった場合には無視される。ストリームに関するエラーおよびファイル終了指示子がリセットされる。

freopen は主に、標準テキスト・ストリーム(stderr, stdin, または stdout)に関連するファイルを変更する際に使用する。これは、fopen関数により返される値を代入できる可変左辺値をこれらの指示子が必要としないからである。

freopen関数はストリームを制御するオブジェクトへのポインタを返す。オープン操作が失敗した場合、freopen は空ポインタを返す。

```
void setbuf(FILE *stream, char *buf);
```

値を返さない点を除き、setbuf関数は setvbuf関数と同じ。setvbuf関数は、*mode* が _IOFBF で、*size* が BUFSIZ で、*mode* が _IONBF の場合 (*buf* が空ポインタの場合) に呼び出される関数である。

```
int setvbuf(FILE *stream, char *buf, int mode size_t size);
```

バッファを入力または出力ファイルに関連付ける。setvbuf関数は、*stream* が示すストリームがオープンしたファイルに関連付けられた後で、その他の操作をストリーム上で実行する前に実行することができる。*mode*実引数は、*stream* をバッファ内に格納する方法を決定する。

- IOFBF は入出力を完全にバッファ内に格納する。
- IOLBF は入出力を行バッファ内に格納する。

- IONBF は入出力をバッファ内に格納しない。

buf が空ポインタ以外の場合は `setvbuf` 関数で割り当てたバッファの代わりに、示される配列を使用することができる。配列のサイズは *size* で指定する。ある時点での配列の内容は不定である。`setvbuf` 関数は成功した場合には 0 を返し、*mode* に無効な値を指定した場合、または要求を受け入れることができない場合には 0 以外の値を返す。

書式付き入出力関数

```
int fprintf(FILE *stream, const char *format, ...);
```

format (書式) が示す文字列の制御に基づき、*stream* が示すストリームへ出力を書き込む。書式は後続の実引数が出力用に変換される方法を指定する。実引数の数が書式より少ない場合の動作結果は定義されていない。書式をすべて使用しても実引数が残っている場合には余分な実引数は評価されるが、その他の点では無視される。`fprintf` 関数は書式文字列の終わりに達すると戻る。`fprintf` 関数は転送された文字の数を返し、出力エラーが起きた場合は負の値を返す。

詳細については、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

```
int fscanf(FILE *stream, const char *format, ...);
```

format が示す文字列の制御に基づき、*stream* が示すストリームから入力を読み込む。書式は、入力可能なシーケンスと、これらのシーケンスが代入のために変換される方法を指定し、後続の実引数をオブジェクトへのポインタとして使用して、変換された入力を受け取る。実引数の数が書式より少ない場合の動作結果は定義されていない。書式をすべて使用しても実引数が残っている場合には余分な実引数は評価されるが、その他の点では無視される。

`fscanf` 関数は、変換の前に入力の失敗が起こった場合に EOF マクロの値を返す。それ以外の場合には、`fscanf` は代入した入力項目の数を返す。この数は入力された数より少ないことがあり、一致しない場合には 0 になることもある。

詳細については、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

```
int printf(const char *format, ...);
```

printf が標準出力ストリーム (stdout) に書式付き出力を書き込む点を除き、fprintf関数と等価である。

```
int scanf(const char *format, ...);
```

scanfが標準入力ストリーム (stdin) から書式付き入力を読み込む点を除き、fscanf関数と等価である。

```
int sprintf(char *s, const char *format, ...);
```

s実引数が、生成する出力の書込み先としてストリームではなく配列を指定する点を除き、fprintf関数と等価である。書き込まれた文字の終わりにはヌル文字が書き込まれる。重複するオブジェクト間でコピーが行われた場合の動作は定義されていない。sprintf関数は配列へ書き込まれた文字の数を返す。終了ヌル文字はカウントされない。

```
int sscanf(const char *s, const char *format, ...);
```

s実引数が、ストリームではなく配列から入力を読み込むことを指定する点を除き、fscanf関数と等価である。文字列の終わりに達することは、fscanf関数がファイルの終わりに達することと同じことを示す。重複するオブジェクト間でコピーが行われた場合の動作結果は定義されていない。

```
#include <stdarg.h>
```

```
int vfprintf(FILE *stream, const char *format, va_list arg);
```

ffprintf関数と等価ですが、arg に置き換えられた可変個の実引数並びは、va_startマクロ (および場合により後続の va_arg呼出し) によって初期化されていなければならない。vfprintf関数は va_endマクロを呼び出さない。

```
#include <stdarg.h>
```

```
int vprintf(const char *format, va_list arg);
```

printf関数と等価ですが、arg に置き換えられた可変個の実引数並びは、va_startマクロ (および場合により後続の va_arg呼出し) によって初期化されていなければならない。vprintf関数は va_endマクロを呼び出さない。

```
#include <stdarg.h>
int vsprintf(char *s, const char *format, va_list arg);
```

`sprintf`関数と等価ですが、*arg* に置き換えられた可変個の実引数並びは、`va_start`マクロ (および場合により後続の `va_arg`呼出し) によって初期化されていなければならない。`vsprintf`関数は `va_end`マクロを呼び出さない。

文字入出力関数

```
int fgetc(FILE *stream);
```

次の文字 (存在する場合) を、*stream* が示す入力ストリームから `int` へ変換された `unsigned char` として返し、ストリームの関連するファイル位置指示子 (定義されている場合) を進める。ストリームがファイルの終わりである場合にはストリームのファイル終了指示子がセットされ、`fgetc` は EOF を返す。読み込みエラーが起きた場合はエラー指示子がセットされ、`fgetc` は EOF を返す。

```
char *fgets(char *s, int n, FILE *stream);
```

stream が示すストリームから、*n* で指定した文字数よりも 1 つ以上少ない文字数を、*s* が示す配列へ読み込む。改行文字 (これは保存される) を読み込んだ後、またはファイルの終わりに達した後は、追加の文字の読み込みはない。配列へ読み込まれた最後の文字の直後にヌル文字が書き込まれない。

成功した場合、`fgets`関数は *s* を返す。ファイルの終わりに達し、文字が配列へ読み込まれなかった場合には配列の内容は変更されず、空ポインタが返される。操作中に読み込みエラーが起きた場合は配列の内容は不定になり、空ポインタが返される。

```
int fputc(int c, FILE *stream);
```

文字 *c* (`unsigned char` へ変換される) を、*stream* が示す出力ストリームに関連するファイル位置指示子 (定義されている場合) が指示する位置に書き込み、必要に応じて指示子を進める。ファイルが位置決め要求をサポートできない場合、またはストリームが追加モードでオープンされた場合には、出力ストリームへ文字が追加される。`fputc`関数は書き込まれた文字を返す。書き込みエラーが起きた場合はストリームのエラー指示子がセットされ、`fputc` は EOF を返す。


```
int fputs(const char *s, FILE *stream);
```

s が示す文字列を *stream* が示すストリームへ書き込む。終了ヌル文字は書き込まれない。

書き込みエラーが起きた場合、fputs関数は EOF を返し、それ以外の場合は負以外の値を返す。

```
int getc(FILE *stream);
```

マクロとして実現されている場合、*stream* を複数回評価される可能性がある点を除き、fgetc関数と等価である。このため、実引数は副作用がある式であってはならない。

```
int getchar(void);
```

stdin実引数を持つ getc関数と等価である。

```
char *gets(char *s);
```

stdin が示す入力ストリームから *s* が示す配列へ、ファイルの終わりに達するか改行文字が読み込まれるまで文字を読み込む。改行文字は破棄され、配列へ読み込まれた最後の文字の直後にヌル文字が書き込まれる。

成功した場合、fgets関数は *s* を返す。ファイルの終わりに達し、文字が配列へ読み込まれなかった場合には配列の内容は変更されず、空ポインタが返される。操作中に読み込みエラーが起きた場合は配列の内容は不定になり、空ポインタが返される。

```
int putc(int c, FILE *stream);
```

マクロとして実現されている場合、*stream* を複数回評価される可能性がある点を除き、fputc関数と等価である。このため、実引数は副作用がある式であってはならない。

```
int putchar(int c);
```

stdout第2実引数を持つ putc関数と等価である。

```
int puts(const char s);
```

s が示す文字列を `stdout` が示すストリームへ書き込み、出力に改行文字を追加する。終了ヌル文字は書き込まれない。書き込みエラーが起きた場合、`puts`関数は EOF を返し、それ以外の場合には負以外の値を返す。

```
int ungetc(int c, FILE *stream);
```

文字 *c* (`unsigned char` に変換される) を、*stream* が示す入力ストリームへプッシュ・バックし、ストリームをその文字の前の位置のままにしておく。プッシュ・バックされた文字は、このストリーム上の次の読み込みによって、プッシュ・バックされたのとは逆順で返される。このストリームに対するファイル位置決め関数 (`fseek`, `fsetpos`, または `rewind`) の呼出しに成功すると、プッシュ・バックされた文字は破棄される。

ファイル上に前の処理が存在しない場合でも、1 回のプッシュ・バックが保証される。`ungetc`関数は変換済みのプッシュ・バックされた文字を返すか、または操作が失敗した場合には EOF を返す。

直接入出力関数

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

stream が示すストリームから、サイズが *size* の要素を最大で *nmemb*個まで *ptr* が示す配列へ読み込む。ストリームのファイル位置指示子 (定義されている場合) は、読み込みに成功した文字数だけ進む。エラーが生じた場合、ストリームのファイル位置指示子の値は不定になる。要素の一部が読み込まれた場合、その値は不定になる。

`fread`関数は読み込みに成功した要素の数を返すが、読み込みエラーまたはファイルの終わりを検出した場合は *nmemb* より少なくなる。*size* または *nmemb* が 0 の場合には `fread` は 0 を返し、配列の内容とストリームの状態は変更されない。

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

ptr が示す配列から、サイズが *size* の要素を最大で *nmemb*個まで、*stream* が示すストリームへ書き込む。ストリームのファイル位置指示子 (定義されている場合) は、書き込みに成功した文字数だけ進む。エラーが生じた場合、ストリームのファイル位置指示子の値は不定になる。

`fwrite`関数は書込みに成功した要素の数を返すが、書込みエラーが見つかった場合にのみ `nmemb` より少なくなる。

ファイル位置決め関数

```
int fgetpos(FILE *stream, fpos_t *pos);
```

`stream` が示すストリームのファイル位置指示子の現在値を、`pos` が示すオブジェクトへ格納する。その値には未指定の情報も含まれており、それは `fgetpos` 関数を呼び出したときに、`fsetpos`関数がストリームをその位置に返すために使用する。

成功した場合、`fgetpos`関数は 0 を返す。失敗した場合、`fgetpos` は 0 以外を返し、処理系定義の正の値を `errno` に格納する。

```
int fseek(FILE *stream, long int offset, int whence);
```

ファイル位置指示子を、`stream` が示すストリーム内の指定されたバイト・オフセットにセットする。

バイナリ・ストリームの場合、ファイルの先頭からの文字数で新しい位置を取得するには `offset` を `whence` で指定する位置に追加する。この位置は次のいずれかである。

- `whence` が `SEEK_SET` の場合、ファイルの先頭
- `whence` が `SEEK_CUR` の場合、ファイル位置指示子の現在値
- `whence` が `SEEK_END` の場合、ファイルの終わり

テキスト・ストリームの場合、`offset` は 0 になるか、または同じストリームに対する `ftell`関数への以前の呼出しによって返された値になる。`whence` は `SEEK_SET` になる。

`fseek` への呼出しが成功すると、ストリームのファイル終了指示子はリセットされ、同じストリーム上での `ungetc`関数の効果は取り消される。`fseek` を呼び出した後、更新ストリーム上での次の操作は入力または出力になる。要求を満足させることができない場合にのみ、`fseek`関数は 0 以外の値を返す。

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

pos が示すオブジェクトの値に従って、*stream* が示すストリームのファイル位置指示子をセットする。このオブジェクト値は、同じストリーム上での *fgetpos* 関数への以前の呼出しから得られる。

fsetpos への呼出しが成功するとストリームのファイル終了指示子はリセットされ、同じストリーム上での *ungetc* 関数の効果は取り消される。*fsetpos* を呼び出した後、更新ストリーム上での次の操作は入力または出力になる。

成功した場合、*fsetpos* 関数は 0 を返す。失敗した場合、*fsetpos* は 0 以外を返し、処理系定義の正の値を *errno* に格納する。

```
long int ftell(FILE *stream);
```

stream が示すストリームのファイル位置指示子の現在値を取得する。バイナリ・ストリームの場合、値はファイルの先頭からの文字数である。テキスト・ストリームの場合、そのファイル位置指示子には未指定の情報も含まれており、それは *ftell* 関数を呼び出したときに、*fseek* 関数がストリームのファイル位置指示子をその位置に返すために使用する。この 2 つの返却値の差は、必ずしも書込みまたは読み込みが行われた文字数の意味ある尺度ではない。

成功した場合、*ftell* 関数はストリームのファイル位置指示子の現在値を返す。失敗した場合、*ftell* は -1L を返し、処理系定義の正の値を *errno* に格納する。

```
void rewind(FILE *stream);
```

stream が示すストリームのファイル位置指示子をファイルの先頭にセットする。ストリームのエラー指示子もリセットされる点を除き、次の関数と等価である。

```
(void)fseek(stream, 0L, SEEK_SET)
```

rewind 関数は値を返さない。

エラー処理関数

```
void clearerr(FILE *stream);
```

stream が示すストリームのファイル終了指示子とエラー指示子をリセットする。clearerr関数は値を返さない。

```
int feof(FILE *stream);
```

stream が示すストリームのファイル終了指示子を検査する。feof関数はファイル終了指示子が *stream* に対してセットされた場合にのみ、0 以外の値を返す。

```
int ferror(FILE *stream);
```

stream が示すストリームのファイル終了指示子を検査する。ファイル終了指示子が *stream* に対してセットされた場合にのみ、ferror関数は0 以外の値を返す。

```
void perror(const char *s);
```

errno整数式のエラー番号をエラー・メッセージにマップする。次に示す文字列を標準エラー・ストリームへ書き込む。

1. *s* が示す文字列とその後にコロン(:)と空白(*s* が空ポインタでなく、*s* が示す文字がヌル文字以外の場合)
2. 適切なエラー・メッセージ文字列と改行文字

エラー・メッセージ文字列の内容は errno実引数を持つ strerror関数が返すものと同じであり、処理系定義である。perror関数は値を返さない。

9.14 汎用ユーティリティ (<stdlib.h>)

<stdlib.h>ヘッダ・ファイルは汎用の4つの型と複数の関数を宣言し、複数のマクロを定義します。関数は文字列変換、乱数生成、検索とソート、メモリ管理、およびそれに類似したタスクを実行します。

型

size_t

sizeof演算子の結果の符号なし汎整数型。

wchar_t

サポートされるロケール間で指定されている最大拡張文字集合のすべてのメンバに対して、個別のコードで表すことができる値の範囲を持つ汎整数型。

div_t

div関数によって返される値の型である構造体型。

ldiv_t

ldiv関数によって返される値の型である構造体型。

マクロ

NULL

処理系定義の空ポインタ定数に展開する。

EXIT_FAILURE/EXIT_SUCCESS

exit関数の実引数として使用して、それぞれ不成功または成功の終了状態をホスト環境に返すための汎整数式に展開する。これらのマクロは、main関数からの返却値としても有用である。

RAND_MAX

rand関数から返される最大値を持つ汎整数定数式に展開する。

MB_CUR_MAX

現在のロケール (LC_TYPE カテゴリ) で指定されている拡張文字集合の多バイト文字中の最大バイト数の値を持つ正の汎整数式に展開する。この値は、常に MB_LEN_MAX 以下である。

文字列変換関数

`double atof(const char *nptr);`

nptr が示す文字列を `double` 表現に変換し、変換された値を返す。エラー発生時の動作を除き、この関数は次に示すものと等価である。

`strtod(nptr, (char **)NULL)`

`int atoi(const char *nptr);`

nptr が示す文字列を `int` 表現に変換し、変換された値を返す。エラー発生時の動作を除き、この関数は次に示すものと等価である。

`(int)strtol(nptr, (char **)NULL, 10)`

`long int atol(const char *nptr);`

nptr が示す文字列を `long int` 表現に変換し、変換された値を返す。エラー発生時の動作を除き、この関数は次に示すものと等価である。

`strtol(nptr, (char **)NULL, 10)`

`double strtod(const char *nptr, char **endptr);`

nptr が示す文字列を `double` 表現に変換する。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

`long int strtol(const char *nptr, char **endptr, int base);`

nptr が示す文字列を `long int` 表現に変換する。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

`unsigned long int strtoul(const char *nptr, char **endptr, int base);`

nptr が示す文字列を `unsigned long int` 表現に変換する。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

擬似乱数列発生関数

```
int rand(void);
```

擬似乱整数列を 0 から `RAND_MAX` の範囲で返す。

```
void srand(unsigned int seed);
```

実引数を、後続の `rand` 呼出しによって返される新しい擬似乱整数列のシードとして使用する。その後に `srand` が同じシード値で呼び出された場合には、同じ擬似乱整数列が繰り返される。`srand` への呼出しを行う前に `rand` が呼び出された場合、生成される列は `srand` をシード値 1 で最初に呼び出したときと同じ。`srand` 関数は値を返さない。

メモリ管理関数

```
void *calloc(size_t nmemb, size_t size);
```

`nmemb` 項目の配列に対して、それぞれサイズが `size` でメモリ領域を割り当てる。領域はすべてビット 0 に初期化される。割当て不可能であった場合、`calloc` 関数は空ポインタを返し、それ以外の場合には割り当てた領域へのポインタを返す。

```
void free(void *ptr);
```

`ptr` が示すメモリ領域の割当てを解除する。これは `calloc`、`malloc`、または `realloc` によって以前に割り当てられたメモリ領域である。`ptr` が空の場合には割当ての解除は行われない。値も返されない。

```
void *malloc(size_t size);
```

サイズが `size` のオブジェクトに対して、メモリの連続領域を割り当てる。領域は初期化されない。この関数は割り当てた領域へポインタを返すが、割当て不可能の場合は空ポインタを返す。

```
void *realloc(void *ptr, size_t size);
```

`ptr` が示す領域のサイズを `size` で指定されたバイト数に変更する。`ptr` が空の場合、`realloc` の動作は `malloc` と同一になる。領域の内容は、旧サイズと新

サイズのいずれか小さいサイズまでは変更されない。サイズ変更できなかった場合、この関数は空ポインタを返し、それ以外の場合には移送した再割当て済みの領域へのポインタを返す。

環境との連絡

```
void abort(void);
```

SIGABRTシグナルが見つかり、シグナル・ハンドラが返らない限りプログラムの異常終了が起こる。abort関数は呼び出す側に値を返すことはできない。

```
int atexit(void (*func)(void));
```

func が示す関数が、プログラムの正常終了時に実引数なしで呼び出されるように登録する。最大で 32 個までの関数を登録できる。登録が成功した場合、atexit関数は 0 を返す。それ以外の場合は 0 以外を返す。

```
void exit(int status);
```

プログラムの正常終了が起こる。プログラムが `exit` の呼出しを複数回実行した場合の動作結果は定義されていない。実行時には次のことが起こる。

1. atexit によって登録されたすべての関数が、登録と逆順で呼び出される。
2. オープンしているすべての出力ストリームがフラッシュされ、オープンしているすべてのストリームがクローズされ、`tmpfile` によって作成されたすべてのファイルが削除される。
3. 制御がホスト環境へ戻される。*status* の値は `errno` 値に対応する。
 - *status* 値が 0 または `EXIT_SUCCESS` の場合は、成功終了状態が返される。
 - *status* 値が `EXIT_FAILURE` の場合は、不成功終了状態が返される。
 - それ以外の場合には、不成功終了状態が返される。

```
char *getenv(const char *name);
```

ホスト環境によって提供された環境並びを検索する。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

```
int *system(const char *string);
```

string が示す文字列を、コマンド・プロセッサによる実行のためにホスト環境に引き渡す。空ポインタを指定すると、コマンド・プロセッサが存在するかどうかを問い合わせることができる。実引数が空ポインタの場合、*system*関数はコマンド・プロセッサが使用可能であれば0以外を返し、使用可能でなければ0を返す。実引数が空ポインタ以外の場合、返却値はコマンド・プロセッサが返した状態になり、コマンド・プロセッサが使用可能ではない場合には0になる。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

検索およびソート・ユーティリティ

```
void *bsearch(const void *key, const void *base,  
             size_t nmemb, size_t size, int (*compar)  
             (const void *, const void *));
```

*nmemb*オブジェクトの配列を検索して、*key*が示すオブジェクトと一致する要素を求める。配列の先頭要素は *base* が示す。各要素のサイズは *size* によって指定される。

ユーザは最初に、*compar* が示す関数に従って配列を昇順にソートしなければならない。*bsearch*関数は *compar* が示す指定の比較関数を、比較するオブジェクト (*key*オブジェクトと配列要素) を示す2つの実引数を指定して呼び出す。比較関数は次の値を返す。

- 第1実引数 < 第2実引数の場合には、< 0 の整数
- 第1実引数 > 第2実引数の場合には、> 0 の整数
- 第1実引数 = 第2実引数の場合には、= 0 の整数

*bsearch*関数は配列の一致した要素へのポインタを返し、一致する要素が見つからない場合には空ポインタを返す。

```
void qsort(void *base, size_t nmemb,  
           size_t size, int (*compar) (const void *,  
           const void *));
```

*nmemb*オブジェクトの配列を同じ場所でソートする。配列の先頭要素は *base* が示す。各要素のサイズは *size* によって指定される。

配列の内容は、`compar` が示す比較関数に従って昇順でソートされる。比較関数は比較されるオブジェクトを示す 2 つの実引数を指定して呼び出される。比較関数は次の値を返す。

- 第 1 実引数 < 第 2 実引数の場合には、< 0 の整数
- 第 1 実引数 > 第 2 実引数の場合には、> 0 の整数
- 第 1 実引数 = 第 2 実引数の場合には、= 0 の整数

比較した 2 つの要素が等しい場合には、ソートされた配列の順序は未指定になる。

`qsort`関数は値を返さない。

整数算術関数

```
int abs(int j);
```

j 整数の絶対値を返す。

```
div_t div(int numer, int denom);
```

numer の *denom* による除算の商と剰余を計算する。`div`関数は商と剰余を含む `div_t`型の構造体を返す。

```
int quot;    /* quotient */
int rem;     /* remainder */
```

```
long int labs(long int j);
```

j ロング整数の絶対値を返す。

```
ldiv_t ldiv(long int numer, long int denom);
```

`div`関数と類似していますが、返された構造体 (`ldiv_t`型) の実引数とメンバがすべて `long int`型である点だけが異なる。

多バイト文字関数

```
int mblen(const char *s, size_t n);
```

s が空ポインタ以外の場合、*mblen* は *s* が示す多バイト文字のバイト数を数える。*mblen*関数は *mbtowc* のシフト状態が影響を受けない点を除き、次の関数と等価である。

```
mbtowc((wchar_t *)0, s, n);
```

s が空ポインタの場合、*mblen*関数は多バイト文字コード化がロケールに依存するコード化である場合には 0 以外を返し、それ以外の場合には 0 を返す。

s が空ポインタ以外の場合、*mblen*関数は次の値のいずれか 1 つを返す。

- *s* がヌル文字を示す場合には 0
- 次の *n*バイト以下のバイトが有効な多バイト文字を形成する場合には、多バイト文字のバイト数
- 次の *n*バイト以下のバイトが有効な多バイト文字を形成しない場合には、-1

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

s が空ポインタ以外の場合、*mbtowc* は *s* が示す多バイト文字のバイト数を決める。次に、この多バイト文字に対応する *wchar_t*型の値のコードを設定する(ヌル文字に対応するコードの値は 0)。多バイト文字が有効であり、*pwc* が空ポインタ以外の場合には、*mbtowc* は *pwc* が示すオブジェクトにコードを格納する。*s* が示す配列の *n*バイトまでが検査される。

s が空ポインタの場合、*mbtowc*関数は多バイト文字コード化がロケールに依存するコード化である場合には 0 以外を返し、それ以外の場合には 0 を返す。

s が空ポインタ以外の場合、*mbtowc*関数は次の値のいずれか 1 つを返す。

- *s* がヌル文字を示す場合には、0
- 次の *n*バイト以下のバイトが有効な多バイト文字を形成する場合には、変換された多バイト文字のバイト数
- 次の *n*バイト以下のバイトが有効な多バイト文字を形成しない場合には、-1

```
int wctomb(char *s, wchar_t wchar);
```

値が *wchar* のコードに対応する多バイト文字を表現するために必要なバイト数を、シフト状態の変更を含めて判断する。その後この関数は *s* が空ポインタ以外の場合には、*s* が示す配列オブジェクトに多バイト文字表現を格納する。MB_CUR_MAX文字まで格納される。*wchar* の値が 0 の場合には、wctomb関数は初期シフト状態のままである。

s が空ポインタの場合、wctomb関数は多バイト文字コード化がロケールに依存するコード化である場合には 0 以外の値を返し、それ以外の場合には 0 を返す。

s が空ポインタ以外の場合、wctomb関数は次の値のいずれか 1 つを返す。

- *wchar* の値が有効な多バイト文字に対応しない場合には、-1
- *wchar* の値に対応する多バイト文字のバイト数

多バイト文字列関数

```
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

初期シフト状態で始まる多バイト文字列を *s* が示す配列から対応するコード列に変換し、*n*以下のコードを *pwcs* が示す配列へ格納する。ヌル文字はコード値 0 に変換される。ヌル文字に続く多バイト文字は検査も変換も行われぬ。各多バイト文字は、mbtowc のシフト状態が影響を受けない点以外は mbtowc を呼び出した場合と同様に変換される。

無効な多バイト文字を検出した場合、mbstowcs関数は (size_t) - 1 を返す。それ以外の場合には変更された配列要素の数を返すが、終了の 0 コード (存在する場合) は含まない。

```
size_t wctombs(char *s, const wchar_t *pwcs, size_t n);
```

多バイト文字に対応するコード列を、*pwcs* が示す配列から初期シフト状態で始まる多バイト文字列へ変換し、これらの多バイト文字を *s* が示す配列へ格納する。1つの多バイト文字の合計が *n* バイトを超えた場合、またはヌル文字が格納された場合、変換は停止する。

各コードは、wctomb のシフト状態が影響を受けない点以外は wctomb を呼び出した場合と同様に変換される。

有効な多バイト文字に対応しないコードを検出した場合、wcstombs関数は (size_t) - 1 を返す。それ以外の場合には変更されたバイト数を返すが、終了ヌル文字 (存在する場合) は含まない。

9.15 文字列処理 (<string.h>)

<string.h>ヘッダ・ファイルは1つの型と複数の関数を宣言し、他のオブジェクトが文字配列として扱う文字配列を処理するために必要な1つのマクロを定義します。

宣言される文字列関数は2種類あります。第1の関数は名前がstrで始まる関数で、文字配列を処理します。第2の関数は名前がmemで始まる関数で、文字配列として扱われるその他のオブジェクトを処理します。重複するオブジェクト間でコピーが行われた場合、memmove以外の関数の動作結果は定義されていません。

型

size_t

sizeof演算子の結果の符号なし汎整数型。

マクロ

NULL

処理系定義の空ポインタ定数に展開する。

関数

void *memcpy(void *s1, const void *s2, size_t n);

n文字をs2が示すオブジェクトからs1が示すオブジェクトへコピーする。関数はs1を返す。

void *memmove(void *s1, const void *s2, size_t n);

n文字分だけ、s2が示すオブジェクトからs1が示すオブジェクトへコピーする。コピーは、最初にs2が示すオブジェクトからn文字分がs1およびs2が示すオブジェクトと重複しないn文字の一時的配列へコピーされ、次に一時的配列からn文字分がs1が示すオブジェクトへコピーされるのと同様に行われる。memmove関数はs1を返す。

```
void *memchr(const void *s, int c, size_t n);
```

c (符号なし char へ変換される) が最初に使用された部分を, *s* が示すオブジェクトの符号なし文字の先頭の *n*文字で検索する。memchr関数は検索された文字へのポインタを返し, 文字が見つからなかった場合には空ポインタを返す。

```
int memcmp(const void *s1, const void *s2, size_t n);
```

s1 が示すオブジェクトの先頭の *n*文字を, *s2* が示すオブジェクトの先頭の *n*文字と比較する。memcmp関数は *s1* が示すオブジェクトが *s2* が示すオブジェクトより小さいか, 等しいか, または大きいかに応じて, それぞれ0より小さい, 等しい, または大きい整数を返す。

```
void *memset(void *s, int c, size_t n);
```

c (符号なし char へ変換される) の値を, *s* が示す先頭の *n*文字へコピーする。関数は *s* を返す。

```
char *strcpy(char *s1, const char *s2);
```

s2 が示す文字列 (終了ヌル文字を含む) を, *s1* が示す文字列へコピーする。strcpy関数は *s1* を返す。

```
char *strncpy(char *s1, const char *s2, size_t n);
```

s2 が示す文字列から終了ヌル文字の前まで, *n*以下の文字を *s1* が示す文字列へコピーする。strncpy関数は *s1* を返す。*s2* が示す文字列が *n*文字より少ない場合, strncpy はコピーにヌル文字を埋め込む。

```
char *strcat(char *s1, const char *s2);
```

s2 が示す文字列 (終了ヌル文字を含む) のコピーを *s1* が示す文字列の終わりに追加する。strcat関数は *s1* を返す。*s2* の先頭文字は *s1* のヌル文字を重ね書きする。

```
char *strncat(char *s1, const char *s2, size_t n);
```

s2 が示す文字列 (ヌル文字の前まで) の *n*以下の文字を *s1* が示す文字列へ追加する。strncat関数は *s1* を返す。*s2* の先頭文字は *s1* のヌル文字を重ね書きする。終了ヌル文字が結果へ追加される。

```
int strcmp(const char *s1, const char *s2);
```

s1 が示す文字列を *s2* が示す文字列と比較する。strcmp関数は *s1* が示す文字列が *s2* が示す文字列より小さいか、等しいか、または大きいかに応じて、それぞれ 0 より小さい、等しい、または大きい整数を返す。

```
int strcoll(const char *s1, const char *s2);
```

s1 が示す文字列を *s2* が示す文字列と比較する。両方の文字列とも、現在のロケール (第 9.6 節を参照のこと) の LC_COLLATE カテゴリに合うように解釈される。strcoll関数は両方の文字列が現在のロケールに適切であると解釈された場合に、*s1* が示す文字列が *s2* が示す文字列より小さいか、等しいか、または大きいかに応じて、それぞれ 0 より小さい、等しい、または大きい整数を返す。

```
int strncmp(const char *s1, const char *s2, size_t n);
```

s1 が示す文字列の *n* 以下の文字を、*s2* が示す文字列と比較する。文字列の比較はヌル文字を検出するまで、文字列に相違点が出るまで、または *n* に達するまで行われる。strncmp関数は *s1* が示す文字列が *s2* が示す文字列より小さいか、等しいか、または大きいかに応じて、それぞれ 0 より小さい、等しい、または大きい整数を返す。

```
size_t strxfrm(char *s1, const char *s2, size_t n);
```

s2 が示す文字列を変換して、*s1* が示す配列へ格納する。

この関数についての詳細は、*Compaq C* ライブラリ・ルーチンのマニュアルを参照すること。

```
char *strchr(const char *s, int c);
```

c (char へ変換される) が最初に使用された部分を、*s* が示す文字列の中で検索する。終了ヌル文字は文字列の一部とみなされる。関数は検索された文字へのポインタを返し、文字が見つからなかった場合には空ポインタを返す。

```
size_t strcspn(const char *s1, const char *s2);
```

s2 が示す文字列に見つからない文字のみから構成される、*s1* が示す文字列の最大の初期セグメント長を計算する。strcspn関数はセグメント長を返す。


```
char *strpbrk(const char *s1, const char *s2);
```

s2 に示された文字列の任意の文字が、*s1* に示された文字列中に使用された最初の部分を検索する。関数はその文字へのポインタを返し、*s2* 中の文字が *s1* に見つからなかった場合には空ポインタを返す。

```
char *strrchr(const char *s, int c);
```

c (char に変換される) が、*s* に示された文字列で使用された最後の部分を検索する。終了ヌル文字は文字列の一部とみなされる。関数は検索された文字へのポインタを返し、文字が見つからなかった場合には空ポインタを返す。

```
size_t strspn(const char *s1, const char *s2);
```

s2 が示す文字列の文字のみから構成される、*s1* が示す文字列の最大の初期セグメント長を計算する。strspn関数はセグメント長を返す。

```
char *strstr(const char *s1, const char *s2);
```

s2 に示された文字列の任意の文字列 (終了ヌル文字を除く) が、*s1* に示された文字列中に使用された最初の部分を検索する。strstr関数は検索された文字列へのポインタを返し、文字列が見つからなかった場合には空ポインタを返す。*s2* が長さ 0 の文字列を示している場合には、関数は *s1* を返す。

```
char *strtok(const char *s1, char *s2);
```

s1 が示す文字列をトークンのシーケンスに分割する。各トークンは *s2* が示す文字列の文字で区切られる。strtok() の最初の呼出しで文字をスキップし、*s2* にない先頭文字を探す。関数は、*s1* が示す文字列中で呼出しから呼出しまでの文字の位置を追跡し、呼出しが連続的に行われると関数はこの文字列を通じて動作し、前回の呼出しによって識別されたテキスト・トークンの後に続くテキスト・トークンを識別する。関数は *s2* 中の文字と一致する文字を *s1* 中に見つけると、その *s1* 中の文字をヌル文字に置き換える。strtok関数はトークンの先頭文字へのポインタを返し、トークンがない場合には空ポインタを返す。

```
char *strerror(int errnum);
```

errnum のエラー番号をエラー・メッセージ文字列にマップし、文字列へのポインタを返す。示された文字列をプログラムで変更することはできないが、後続の strerror への呼出しによって重ね書きすることができる。

```
size_t strlen(const char *s);
```

s が示す文字列の長さを計算する。関数は終了ヌル文字に先行する文字数を返す。

9.16 型汎用数学 (<tgmath.h>)

<tgmath.h>ヘッダは、<math.h>ヘッダと<complex.h>ヘッダを含み、いくつかの型汎用マクロを定義しています。

f (float) や l (long double) 接尾語がない、<math.h>および<complex.h>の関数には、対応する実数型がdoubleであるパラメータを1つ以上持っているものがあります。このような関数 (modfを除く) には、対応する型汎用マクロがあります。¹対応する実数型が関数形式内でdoubleのパラメータは、汎用パラメータです。このマクロを使用すると、汎用パラメータの引数で関数の実数型と型ドメインが決定され、関数が呼び出されます。²

注意

絶対値関数 (fabs) の型汎用機能は、本リリースでは複素数型には利用できません。代わりに、各型専用の名前 (cabs, cabsf, cabs1) を使用しなければなりません。

¹ 標準ライブラリの他の関数形式マクロのように、対応する通常関数を利用できるよう、各型汎用マクロを抑制することもできます。

² 選択された関数のパラメータの型と引数の型が合っていない場合、その動作は未定義です。

9.16.1 実数型の決定

マクロを使用すると、汎用パラメータに対応する実数型が次の手順で決定され、その型の関数が呼び出されます。

1. まず、汎用パラメータにlong double型の引数がある場合、その型はlong doubleになります。
2. long double型の引数がなく、汎用パラメータにdouble型の引数または整数型の引数がある場合、その型はdoubleになります。
3. 上記以外の場合、その型はfloatになります。

9.16.2 <math.h>と<complex.h>で同じ名前の接尾語なしの関数

<math.h>内の接尾語のない関数で、<complex.h>内に同じ名前にc接頭語が付いた関数があるものには、<math.h>内の関数と同じ名前の型汎用マクロ (両方の関数用) があります。fabsとcabsに対応する型汎用マクロは、fabsです。このような関数を、次に示します。

<math.h> 関数	<complex.h> 関数	型汎用 マクロ
acos	cacos	acos
asin	casin	asin
atan	catan	atan
acosh	cacosh	acosh
asinh	casinh	asinh
atanh	catanh	atanh
cos	ccos	cos
sin	csin	sin
tan	ctan	tan
cosh	ccosh	cosh
sinh	csinh	sinh
tanh	ctanh	tanh
exp	cexp	exp
log	clog	log
pow	cpow	pow
sqrt	csqrt	sqrt
fabs	cabs	fabs

汎用パラメータに複素数の引数が少なくとも 1 つあれば、マクロを使用すると、複素数関数が呼び出されます。複素数の引数がなければ、マクロを使用すると実数関数が呼び出されます。

9.16.3 対応する c 接頭語関数が<complex.h>にない、<math.h>の接尾語なし関数

<math.h>内の接尾語のない関数のうち、対応する c 接頭語関数が<complex.h>にない関数には、その関数と同じ名前の、対応する型汎用マクロがあります。このような型汎用マクロを、次に示します。

atan2	fma	llround	remainder
cbrt	fmax	log10	remquo
ceil	fmin	log1p	rint
copysign	fmod	log2	round
erf	frexp	logb	scalbn
erfc	hypot	lrint	scalbln
exp2	ilogb	lround	tgamma
expm1	ldexp	nearbyint	trunc
fdim	lgamma	nextafter	
floor	llrint	nexttoward	

汎用パラメータのすべての引数が実数の場合、マクロを使用すると実数関数が呼び出されます。実数でない引数がある場合、マクロの動作は定義されていません。

9.16.4 <math.h>内の関数に対応する c 接頭語付き関数ではない、<complex.h>内の接尾語なし関数

<math.h>内の関数に対応する c 接頭語付き関数ではない、<complex.h>内の接尾語なし関数には、その関数と同じ名前の型汎用マクロがあります。

carg	conj	creal
cimag	cproj	

このマクロを実数または複素数の引数で使用すると、複素数関数が呼び出されます。

9.16.5 例

次のような宣言があるとします。

```
#include <tgmath.h>
int n;
float f;
double d;
long double ld;
float complex fc;
double complex dc;
long double complex ldc;
```

このような宣言があると、型汎用マクロを使用したときに呼び出される関数は、次のとおりとなります。

使用するマクロ	呼び出し
-----	-----
exp(n)	exp(n) , 関数
acosh(f)	acoshf(f)
sin(d)	sin(d) , 関数
atan(ld)	atanl(ld)
log(fc)	clogf(fc)
sqrt(dc)	csqrt(dc)
pow(ldc, f)	cpowl(ldc, f)
remainder(n, n)	remainder(n, n) , 関数
nextafter(d, f)	nextafter(d, f) , 関数
nexttoward(f, ld)	nexttowardf(f, ld)
copysign(n, ld)	copysignl(n, ld)
ceil(fc)	動作は未定義
rint(dc)	動作は未定義
fmax(ldc, ld)	動作は未定義
carg(n)	carg(n) , 関数
cproj(f)	cprojf(f)
creal(d)	creal(d) , 関数
cimag(ld)	cimagl(ld)
cabs(fc)	cabsf(fc)
carg(dc)	carg(dc) , 関数
cproj(ldc)	cprojl(ldc)

9.16.6 虚数引数

複素数引数を使用できる型汎用マクロでは、虚数引数も使用できます。引数が虚数の場合、マクロはその関数に合わせて、型が実数、虚数、または複素数の式に展開されます。引数が虚数の場合、`cos`、`cosh`、`fabs`、`carg`、`cimag`、および`creal`の型は実数になり、`sin`、`tan`、`sinh`、`tanh`、`asin`、`atan`、`asinh`、および`atanh`の型は虚数になり、その他の関数の型は複素数になります。

引数が虚数の場合、型汎用マクロ`cos`、`sin`、`tan`、`cosh`、`sinh`、`tanh`、`asin`、`atan`、`asinh`、`atanh`は、実数関数を使って次の式で規定されます。

```
cos(iy)   = cosh(y)
sin(iy)   = i sinh(y)
tan(iy)   = i tanh(y)
cosh(iy)  = cos(y)
sinh(iy)  = i sin(y)
tanh(iy)  = i tan(y)
asin(iy)  = i asinh(y)
atan(iy)  = i atanh(y)
asinh(iy) = i asin(y)
atanh(iy) = i atan(y)
```

9.17 日付と時刻 (<time.h>)

<time.h>ヘッダ・ファイルは2つのマクロを定義し、時間と日付の情報を処理するための4つの型と複数の関数を宣言します。一部の関数は、時間帯により暦時間と異なることがあるローカル時間を処理します。

型

`size_t`

`sizeof`演算子の結果の符号なし汎整数型。

`clock_t`

`time_t`

時間を表現できる算術型。

struct tm

詳細時間と呼ばれる暦時間の構成要素を保持する。構造体は次のメンバからなる。

```
int tm_sec;      /* seconds after the minute -- [0,61]      */
int tm_min;      /* minutes after the hour -- [0,59]      */
int tm_hour;     /* hours since midnight -- [0,23]      */
int tm_mday;     /* day of the month -- [1,31]      */
int tm_mon;      /* months since January -- [0,11]      */
int tm_year;     /* years since 1900      */
int tm_wday;     /* days since Sunday -- [0,6]      */
int tm_yday;     /* days since January 1 -- [0,365]      */
int tm_isdst;    /* Daylight Saving Time flag -- 0 if
/* DST not in effect; positive if it is;
/* negative if information is not available. */
```

マクロ

NULL

処理系定義の空ポインタ定数に展開する。

CLOCKS_PER_SEC

clock関数によって返される値の1秒当たりの数。

時間変換関数

char *asctime(const struct tm *timeptr);

timeptr が示す構造体内の詳細時間を、次の例の形式の文字列に変換する。

Sat Sep 08 08:10:32 1990\n\0

文字列へのポインタが返される。

char *ctime(const time_t *timer);

timer が示す暦時間を、asctime関数によって生成される形式の文字列に変換する。文字列へのポインタが返される。ctime関数は次の関数と等価である。

asctime(localtime(timer))

```
struct tm *gmtime(const time_t *timer);
```

timer が示す暦時間を、協定世界時 (UTC) として表現される詳細時間に変換する。gmtime関数は詳細時間へのポインタを返すか、または UTC が使用可能ではない場合には空ポインタを返す。

```
struct tm *localtime(const time_t *timer);
```

timer が示す暦時間を、ローカル時間として表現される詳細時間に変換する。localtime関数は詳細時間へのポインタを返す。

```
size_t strftime(char *s, size_t maxsize, const char *format, const struct tm *timeptr);
```

format が示す文字列の制御に基づいて、文字を *s* が示す配列へ格納する。書式文字列は、0 個以上の変換指定子と通常の変換文字からなる。通常の変換文字 (終了文字を含む) はすべて、変更されずに配列へコピーされる。各変換指定子は、表 9-2 に示すように適切な文字に置き換えられる。この文字は、現在のロケールの LC_TIMEカテゴリーと、*timeptr* が示す構造体に含まれている値によって決まる。

表 9-2 strftime 変換指定子

指定子	置き換える値
%a	ロケールの短縮曜日名
%A	ロケールの完全曜日名
%b	ロケールの短縮月名
%B	ロケールの完全月名
%c	ロケールの適切な日付と時刻表現
%d	10 進数としての月の日数 (01 ~ 31)
%H	10 進数としての時刻 (24 時クロック) (00 ~ 23)
%I	10 進数としての時刻 (12 時クロック) (01 ~ 12)
%j	10 進数としての年の日数 (001 ~ 366)

(次ページに続く)

表 9-2 (続き) strftime 変換指定子

指定子	置き換える値
%m	10 進数としての月 (01 ~ 12)
%M	10 進数としての分 (00 ~ 59)
%p	12 時クロックに関連する AM/PM 指定に対するロケールの同値
%S	10 進数としての秒 (00 ~ 61)
%U	10 進数としての年の週数 (最初の日曜日が 1 週目の 1 日目) (00 ~ 53)
%w	10 進数としての曜日 (0[日曜日] ~ 6[土曜日])
%W	10 進数としての年の週数 (最初の月曜日が 1 週目の 1 日目) (00 ~ 53)
%x	ロケールの適切な日付表現
%X	ロケールの適切な時刻表現
%y	10 進数としての世紀なしの年 (00 ~ 99)
%Y	10 進数としての世紀付きの年
%Z	時間帯名または短縮形, あるいは時間帯を判定できない場合には文字なし
%%	%

生成された文字の合計数が, 終了ヌル文字を含めて maxsize 以下の場合には, strftime 関数は s が示す配列へ格納された文字数を終了ヌル文字を含めずに返す。それ以外の場合には 0 が返され, 配列の内容は不定になる。

時間処理関数

```
clock_t clock(void);
```

使用したプロセッサ時間を判断する。clock 関数は, プログラムの起動に関連するイベントの発生以降にそのプログラムが使用したプロセッサ時間を返す。時間を秒数で判断するには, 返却値を CLOCKS_PER_SEC マクロの値で除算する。プロセッサ時間を使用できないか, または表現できない場合, 返される値は (clock_t)-1 になる。プログラムで使用した時間を測定するにはプログラムの起動時に clock 関数を呼び出し, 返却値を次の呼出しの返却値から減算する。

```
double difftime(time_t time1, time_t time0);
```

2 つの暦時間である time1 および time0 の差を, double として秒数で表現した値で返す。

```
time_t mktime(struct tm *timeptr);
```

timeptr が示す構造体内の、ロケール時間として表現された詳細時間を *time*関数が返した値と同じコード化で(つまり、*time_t*型の値で)返して、暦時間値に変換する。暦時間を表現できない場合には、(*time_t*)-1値が返される。

tm_wday および *tm_yday* の時間構成要素の元の値は無視され、その他の構成要素の元の値は前述の *struct_tm* の説明で示した範囲に制限されない。関数が正常に完了すると *tm_wday* および *tm_yday* 構成要素の値は適切に設定され、その他の構成要素は指定された暦時間を表現するように設定されるが、その値は *struct_tm* の説明で示した範囲に制限される。*tm_wday* の最終値は、*tm_mon* および *tm_year* が決まるまで設定されない。

```
time_t time(time_t *timer);
```

現在の暦時間を返す。暦時間が使用できない場合には (*time_t*)-1 が返される。

言語構文一覧

この節では、ANSI C 規格の構文を引用して C 言語の構文についてまとめます。構文カテゴリはボールド体で、リテラル語または文字はモノスペースの非イタリック体で表します。構文カテゴリの後のコロンはその構文の定義を示します。定義が複数ある場合は別の行に表示するか、「次のいずれか 1 つ」というように前置しています。オプションの要素には `opt` を付けています。たとえば、次の行はオプションの式を中括弧で囲んでいます。

`{ 式opt }`

括弧内のセクション番号は、C 言語のこの部分を説明している『American National Standard for Information Systems-Programming Language C』（文書番号: X3.159-1989）のセクションを示しています。

A.1.1 レキシカル文法

A.1.1.1 トークン

トークン: (§3.1)

- キーワード
- 識別子
- 定数
- 文字列リテラル
- 演算子
- 区切り子

前処理トークン: (§3.1)

- ヘッダ名
- 識別子
- 前処理数
- 文字定数
- 文字列リテラル

言語構文一覧

演算子

区切り子

上記以外の各非空白文字

A.1.1.2 キーワード

キーワード: (§3.1.1) 次のいずれか 1 つ

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

_Bool _Complex
(Alpha only)

A.1.1.3 識別子

識別子: (§3.1.2)

非数字識別子

識別子 非数字識別子

識別子 数字

非数字識別子:

非数字

他の処理系定義の文字

非数字: (§3.1.2) 次のいずれか 1 つ

a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z _

数字: (§3.1.2) 次のいずれか 1 つ

0 1 2 3 4 5 6 7 8 9

A.1.1.4 定数

定数: (§3.1.3)

浮動小数点定数
 整数定数
 列挙定数
 文字定数

浮動小数点定数: (§3.1.3.1)

10 進浮動小数点定数
 16 進浮動小数点定数

10 進浮動小数点定数

小数定数 指数部_{opt} 浮動小数点接尾語_{opt}
 数字列 指数部 浮動小数点接尾語_{opt}

16 進浮動小数点定数

16 進接頭語 16 進小数定数 2 進指数部 浮動小数点接尾語_{opt}
 16 進接頭語 16 進数字列 2 進指数部 浮動小数点接尾語_{opt}

小数定数: (§3.1.3.1)

数字列_{opt} . 数字列
 数字列 .

指数部: (§3.1.3.1)

e 符号_{opt} 数字列
 E 符号_{opt} 数字列

符号: (§3.1.3.1) 次のいずれか 1 つ

+ -

数字列: (§3.1.3.1)

数字
 数字列 数字

16 進小数定数:

16 進数字列_{opt} . 16 進数字列
 16 進数字列 .

言語構文一覧

2 進指数部:

p 符号_{opt} 数字列

P 符号_{opt} 数字列

16 進数字列:

16 進数字

16 進数字列 16 進数字

浮動小数点接尾語: (§3.1.3.1) 次のいずれか 1 つ

f l F L

整数定数: (§3.1.3.2)

10進定数 整数接尾語_{opt}

8進定数 整数接尾語_{opt}

16進定数 整数接尾語_{opt}

10 進定数: (§3.1.3.2)

0以外の数字

10進定数 数字

8 進定数: (§3.1.3.2)

0

8進定数 8進数字

16 進定数: (§3.1.3.2)

0x 16進数字

0X 16進数字

16進定数 16進数字

0 以外の数字: (§3.1.3.2) 次のいずれか 1 つ

1 2 3 4 5 6 7 8 9

8 進数字: (§3.1.3.2) 次のいずれか 1 つ

0 1 2 3 4 5 6 7

16 進数字: (§3.1.3.2) 次のいずれか 1 つ

0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

整数接尾語: (§3.1.3.2)

符号なし接尾語 長接尾語_{opt}
長接尾語 符号なし接尾語_{opt}

符号なし接尾語: (§3.1.3.2) 次のいずれか 1 つ

u U

長接尾語: (§3.1.3.2) 次のいずれか 1 つ

l L

列挙定数: (§3.1.3.3)

識別子

文字定数: (§3.1.3.4)

' c文字列 '
L ' c文字列 '

c 文字列: (§3.1.3.4)

c文字
c文字列 c文字

c 文字: (§3.1.3.4)

エスケープ・シーケンス
次のものを除くソース文字集合の任意のメンバ
単一引用符 ('), バックスラッシュ (\), または改行文字

エスケープ・シーケンス: (§3.1.3.4)

単純エスケープ・シーケンス
8進エスケープ・シーケンス
16進エスケープ・シーケンス

単純エスケープ・シーケンス: (§3.1.3.4) 次のいずれか 1 つ

```
\'  \"  \?  \\
\a  \b  \f  \n  \r  \t  \v
```

8 進エスケープ・シーケンス: (§3.1.3.4)

```
\8進数字
\8進数字 8進数字
\8進数字 8進数字 8進数字
```

16 進エスケープ・シーケンス: (§3.1.3.4)

```
\x 16進数字
16進エスケープ・シーケンス 16進数字
```

A.1.1.5 文字列リテラル

文字列リテラル: (§3.1.4)

```
" s文字列opt "
L " s文字列opt "
```

s 文字列: (§3.1.4)

```
s文字
s文字列 s文字
```

s 文字: (§3.1.4)

エスケープ・シーケンス
次のものを除くソース文字集合の任意のメンバ
二重引用符 ("), バックスラッシュ (\), または改行文字

A.1.1.6 演算子

演算子: (§3.1.5) 次のいずれか 1 つ

```
[ ] ( ) . ->
++ -- & * + - ~ ! sizeof
/ % << >> < > <= >= == != ^ | && ||
? :
= *= /= %= += -= <<= >>= &= ^= |=
, # ##
```


A.1.1.7 区切り子

区切り子: (§3.1.6) 次のいずれか 1 つ

[] () { } * , : = ; ... #

A.1.1.8 ヘッダ名

ヘッダ名: (§3.1.7)

< h文字列 >

" q文字列 "

h 文字列: (§3.1.7)

h文字

h文字列 h文字

h 文字: (§3.1.7)

次のものを除くソース文字集合の任意のメンバ
改行文字および >

q 文字列: (§3.1.7)

q文字

q文字列 q文字

q 文字: (§3.1.7)

次のものを除くソース文字集合の任意のメンバ
改行文字および "

A.1.1.9 前処理数

前処理数: (§3.1.8)

数字

. 数字

前処理数 数字

前処理数 非数字

前処理数 e 符号

前処理数 E 符号

前処理数 .

A.1.2 句構造文法

A.1.2.1 式

1 次式: (§3.3.1)

識別子

定数

文字列リテラル

(式)

後置式: (§3.3.2)

1 次式

後置式 [式]

後置式 (実引数式並び_{opt})

後置式 . 識別子

後置式 -> 識別子

後置式 ++

後置式 --

実引数式並び: (§3.3.2)

代入式

実引数式並び , 代入式

単項式: (§3.3.3)

後置式

++ 単項式

-- 単項式

単項演算子 キャスト式

sizeof 単項式

sizeof (型名)

単項演算子: (§3.3.3) 次のいずれか 1 つ

& * + - ~ !

キャスト式: (§3.3.4)

単項式

(型名) キャスト式

乗除式: (§3.3.5)

キャスト式

乗除式 * キャスト式

乗除式 / キャスト式

乗除式 % キャスト式

加減式: (§3.3.6)

乗除式

加減式 + 乗除式

加減式 - 乗除式

シフト式: (§3.3.7)

加減式

シフト式 << 加減式

シフト式 >> 加減式

関係式: (§3.3.8)

シフト式

関係式 < シフト式

関係式 > シフト式

関係式 <= シフト式

関係式 >= シフト式

等価式: (§3.3.9)

関係式

等価式 == 関係式

等価式 != 関係式

AND 式: (§3.3.10)

等価式

AND式 & 等価式

排他的 OR 式: (§3.3.11)

AND式

排他的OR式 \wedge AND式

包含 OR 式: (§3.3.12)

排他的OR式

包含OR式 \mid 排他的OR式

論理 AND 式: (§3.3.13)

包含OR式

論理AND式 $\&\&$ 包含OR式

論理 OR 式: (§3.3.14)

論理AND式

論理OR式 $\mid\mid$ 論理AND式

条件式: (§3.3.15)

論理OR式

論理OR式 ? 式 : 条件式

代入式: (§3.3.16)

条件式

単項式 代入演算子 代入式

代入演算子: (§3.3.16) 次のいずれか 1 つ

$=$ $*=$ $/=$ $\%=$ $+=$ $-=$ $<<=$ $>>=$ $\&=$ $\wedge=$ $\mid=$

式: (§3.3.17)

代入式

式 , 代入式

定数式: (§3.4)

条件式

A.1.2.2 宣言

宣言: (§3.5)

宣言指定子 初期化宣言子並び $_{\text{opt}}$;

宣言指定子: (§3.5)

記憶域クラス指定子 宣言指定子_{opt}
 型指定子 宣言指定子_{opt}
 型修飾子 宣言指定子_{opt}

初期化宣言子並び: (§3.5)

初期化宣言子
 初期化宣言子並び , 初期化宣言子

初期化宣言子: (§3.5)

宣言子
 宣言子 = 初期化子

記憶域クラス指定子: (§3.5.1)

typedef
 extern
 static
 auto
 register

型指定子: (§3.5.2)

void
 char
 short
 int
 long
 float
 double
 signed
 unsigned

 _Bool
 _Complex

(Alpha only)

構造体または共用体指定子
 列挙型指定子
 typedef名

構造体または共用体指定子: (§3.5.2.1)

```
struct または union 識別子opt { 構造体宣言並び }  
struct または union 識別子
```

struct または union: (§3.5.2.1)

```
struct  
union
```

構造体宣言並び: (§3.5.2.1)

```
構造体宣言  
構造体宣言並び 構造体宣言
```

構造体宣言: (§3.5.2.1)

```
指定修飾子並び 構造体宣言子並び ;
```

指定修飾子並び: (§3.5.2.1)

```
型指定子 指定修飾子並びopt  
型修飾子 指定修飾子並びopt
```

構造体宣言子並び: (§3.5.2.1)

```
構造体宣言子  
構造体宣言子並び , 構造体宣言子
```

構造体宣言子: (§3.5.2.1)

```
宣言子  
宣言子opt : 定数式
```

列挙型指定子: (§3.5.2.2)

```
enum 識別子opt { 列挙子並び }  
enum 識別子opt { 列挙子並び, }  
enum 識別子
```

列挙子並び: (§3.5.2.2)

```
列挙子  
列挙子並び , 列挙子
```

列挙子: (§3.5.2.2)

列挙定数

列挙定数 = 定数式

型修飾子: (§3.5.3)

const

volatile

宣言子: (§3.5.4)

ポインタ_{opt} 直接宣言子

直接宣言子: (§3.5.4)

識別子

(宣言子)

直接宣言子 [定数式_{opt}]

直接宣言子

直接宣言子 (識別子並び_{opt})

ポインタ: (§3.5.4)

* 型修飾子並び_{opt}

* 型修飾子並び_{opt} ポインタ

型修飾子並び: (§3.5.4)

型修飾子

型修飾子並び 型修飾子

仮引数型並び: (§3.5.4)

仮引数並び

仮引数並び , ...

仮引数並び: (§3.5.4)

仮引数宣言

仮引数並び , 仮引数宣言

仮引数宣言: (§3.5.4)

宣言指定子 宣言子

宣言指定子 抽象宣言子_{opt}

識別子並び: (§3.5.4)

識別子

識別子並び^{*}, 識別子

型名: (§3.5.5)

指定修飾子並び 抽象宣言子_{opt}

抽象宣言子: (§3.5.5)

ポインタ

ポインタ_{opt} 直接抽象宣言子

直接抽象宣言子: (§3.5.5)

(抽象宣言子)

直接抽象宣言子_{opt} [定数式_{opt}]

直接抽象宣言子_{opt} (仮引数型並び_{opt})

typedef 名: (§3.5.6)

識別子

初期化子: (§3.5.7)

代入式

{ 初期化子並び }

{ 初期化子並び^{*}, }

初期化子並び: (§3.5.7)

初期化子

初期化子並び^{*}, 初期化子

A.1.2.3 文

文: (§3.6)

ラベル付き文

複文

式文

選択文

繰返し文

飛越し文

ラベル付き文: (§3.6.1)

識別子 : 文

case 定数式 : 文

default : 文

複文: (§3.6.2)

{ 宣言並び_{opt} 文並び_{opt} }

宣言並び: (§3.6.2)

宣言

宣言並び 宣言

文並び: (§3.6.2)

文

文並び 文

式文: (§3.6.3)

式_{opt} ;

選択文: (§3.6.4)

if (式) 文

if (式) 文 else 文

switch (式) 文

繰返し文: (§3.6.5)

while (式) 文

do 文 while (式) ;

for (式_{opt} ; 式_{opt} ; 式_{opt}) 文

飛越し文: (§3.6.6)

goto 識別子 ;

continue ;

break ;

return 式_{opt} ;

A.1.2.4 外部定義

翻訳単位: (§3.7)

外部宣言

翻訳単位 外部宣言

外部宣言: (§3.7)

関数定義

宣言

関数定義: (§3.7.1)

宣言指定子_{opt} 宣言子 宣言並び_{opt} 複文

A.1.3 前処理命令

前処理ファイル: (§3.8)

グループ_{opt}

グループ: (§3.8)

グループ部

グループ グループ部

グループ部: (§3.8)

前処理トークン_{opt} 改行

ifセクション

制御行

if セクション: (§3.8.1)

ifグループ elifグループ_{opt} elseグループ_{opt} endif行

if グループ: (§3.8.1)

#if 定数式 改行 グループ_{opt}

#ifdef 識別子 改行 グループ_{opt}

#ifndef 識別子 改行 グループ_{opt}

elif グループ: (§3.8.1)

elifグループ

elifグループ群 elifグループ

elif グループ: (§3.8.1)

#elif 定数式 改行 グループ_{opt}

else グループ: (§3.8.1)

#else 改行 グループ_{opt}

endif 行: (§3.8.1)

#endif 改行

制御行:

#include 前処理トークン 改行 (§3.8.2)

#define 識別子 置換並び 改行 (§3.8.3)

#define 識別子 (識別子並び)_{opt} 置換並び 改行 (§3.8.3)

#undef 識別子 改行 (§3.8.3)

#line 前処理トークン 改行 (§3.8.4)

#error 前処理トークン_{opt} 改行 (§3.8.5)

#pragma 前処理トークン_{opt} 改行 (§3.8.6)

改行 (§3.8.7)

左括弧: (§3.8.3)

空白が先行しない左括弧文字

置換並び: (§3.8.3)

前処理トークン_{opt}

前処理トークン: (§3.8)

前処理トークン

前処理トークン群 前処理トークン

改行: (§3.8)

改行文字

ANSI 規格準拠の概要

*Compaq C*はX3J11 技術委員会が規定し、『American National Standard for Information Systems-Programming Language C』(文書番号:X3.159-1989)に文書化されたプログラミング言語 C の ANSI 規格に準拠しています。*Compaq C*は、ANSI 規格準拠に関する Plum-Hall テストをパスしました。厳密な ANSI C 規格モードでは、*Compaq C*コンパイラは ANSI C 規格のセクション 1.7 準拠に次のように記述されている規格準拠の処理系です。

「規格準拠のホスト処理系は、厳密な規格準拠のプログラムをすべて受け入れるものとする。規格準拠の処理系は、言語の拡張が厳密な規格準拠のプログラムの動作を変えないという条件で(追加のライブラリ関数を含めて)拡張機能を持つことができる。」

ANSI C 規格では、厳密な規格準拠のプログラムを次のように定義しています。

「厳密な規格準拠のプログラムは、この規格に規定された言語およびライブラリの機能だけを使用するものとする。同プログラムは未指定、未定義、または処理系定義の動作に依存する出力を生成してはならず、最小の処理系の限界を超えてはならない。」

「処理系には、すべての処理系定義の特性および拡張を定義したドキュメントが伴うものとする。」

ANSI C 規格はほとんどの言語定義と同様に、1つの処理系で利用可能な C 言語の定義全体を含むものではありません。弊社が現在サポートしている C の処理系には、ANSI C 規格では定義されていない多数の機能が組み込まれています。

この節では、コンパイラの機能を ANSI C 規格の概要を反映する形式で説明します。各見出しの後ろの括弧内に、関連する ANSI C 規格のセクション番号を示して

います。説明に ANSI C 規格の見出しが省略されている場合は、*Compaq C*が規格に厳密に準拠しており、拡張または処理系定義の動作がないことを示します。

以降の各節には、*Compaq C*言語の拡張部分と処理系定義の部分だけを記述しています。この節は、ANSI C 規格とともに C 言語の*Compaq C*処理系を完全に規定します。この付録では、ANSI C 規格を単に「規格」と呼びます。

B.1 診断 (§2.1.1.3)

診断メッセージは、規格に規定されている構文規則または制約の最初の違反に対して生成されます。それ以降の違反は、以前の違反によって隠されない場合のみ報告されます。

B.2 ホスト環境 (§2.1.2.2)

`envp` を含めて `main()` の実引数の意味は、プログラミング環境によって決まります。`main()` の実引数についての詳細は、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

B.3 多バイト文字 (§2.2.1.2)

多バイト文字のコード化に使用するシフト状態は、ローカル・システムで利用可能な変換テーブルに依存します。ローカル・システムの変換テーブルによりサポートされている場合は、特定の文字集合はその言語によりサポートされます。

B.4 エスケープ・シーケンス (§2.2.2)

ソース文字集合の文字定数または文字列リテラル内の要素は、実行文字集合の要素へ直接マップされます。規格に定義されているもの以外のエスケープ・シーケンスは、警告の診断メッセージが出されて、バックスラッシュは無視されるため、文字定数または文字列リテラルの値は、バックスラッシュが存在しない場合と同様になります。

B.5 翻訳限界 (§2.2.4.1)

マシン・アーキテクチャとオペレーティング・システムに差異があるため、翻訳限界はプラットフォームによって異なります。それ以外において翻訳限界は存在しません。

次の一覧は、*Compaq C*で設定している限界のみを示しています。規格にあっても次の一覧には存在しない翻訳限界は、*Compaq C*では設定されていないものです。

- 内部識別子またはマクロ名は 32,767 文字
- 論理ソース行または物理ソース行は 32,767 文字
- 文字列リテラルの表現は 32,767 バイト (この限界は、連結の結果生成された文字列リテラルには適用されない)

*Tru64 UNIX*システムの場合

- 外部識別子の有効先頭文字は 1023 文字。このような識別子が切り捨てられた場合、警告が出される。

*OpenVMS*システムの場合

- 外部識別子の有効先頭文字は 31 文字。このような識別子が切り捨てられた場合、警告が出される。
- 関数の実引数または仮引数は 253 個。
- 関数の実引数並びは 1012 バイト。

B.6 数量的限界 (§2.2.4.2)

*Compaq C*の数量的限界は、*limits.h* および *float.h* のヘッダ・ファイルに定義されています。これらのヘッダ・ファイルには、以下の記述に合わせて設定された処理系定義の値が入っています。

- 実行文字集合の 1 文字は 8 ビットです。

- char型の値の表現とセットは signed char型の場合と同じです。コマンド行オプションを使用して、この同値を signed char から unsigned char に変更することができます。
- OpenVMSシステム上では、int および signed int型の値の表現とセットは long型 (32 ビット) の場合と同じです。
- OpenVMSシステム上では、unsigned int型の値の表現とセットは unsigned long型 (32 ビット) の場合と同じです。
- Tru64 UNIXシステム上では、long intおよびunsigned long int型は 64 ビットであり、intおよびunsigned int型は 32 ビットです。
- long double型の値の表現とセットは double型 (64 ビット) の場合と同じです。

ここに記述されていない限界については、規格に示されています。

B.7 キーワード (§3.1.1)

キーワード `__inline`、`__unaligned`、および `__restrict` は、*OpenVMS Alpha* システムおよび *Tru64 UNIX* システム上でサポートされています。

VAX Cのキーワードはすべて、VAX Cモードでサポートされます。これらのキーワードには次のものがあります。

- `_align`
- `globaldef`
- `globalref`
- `globalvalue`
- `noshare`
- `readonly`
- `variant_struct`
- `variant_union`

次のキーワードは *Tru64 UNIX* システム上では受け入れられますが、警告が出されます。

- `_align`
- `noshare`
- `readonly`

Tru64 UNIX システムでは、`globaldef` 宣言と初期化された `globalvalue` 宣言は外部宣言として扱われます。`globalref` 宣言と初期化されていない `globalvalue` 宣言は、`extern` を宣言した場合と同様に扱われます。

注意

MAIN PROGRAM オプションも、OpenVMS システムでは VAX C 互換性オプションで利用できます。

B.8 識別子 (§3.1.2)

識別子にはドル記号 (\$) を含めることができます。ただし、厳密な ANSI モードでは警告が出されます。

Tru64 UNIX システムにおいて外部結合を持つ識別子を指定する場合、大文字と小文字は識別されます。

OpenVMS システムでは、省略時の設定により外部結合を持つ識別子名はすべて大文字に変換されますが、これはコマンド行オプションで制御することができます。

B.9 識別子の結合 (§3.1.2.2)

1 つの翻訳単位内で同じ識別子が内部結合と外部結合の両方で現れた場合は、エラーが報告されます。

B.10 型 (§3.1.2.5)

`char`型と `signed char`型は同じ値の表現とセットを持ちます。ただし、`unsigned` のコンパイル時オプションを指定した場合には、`char`型と `unsigned char`型が同じ値の表現とセットを持ちます。

B.11 整数定数 (§3.1.3.2)

コモン C および VAX C モードでは、数字の 8 と 9 は有効な 8 進数字として許されますが、警告メッセージが出されます。

B.12 文字定数 (§3.1.3.4)

2 つ以上の文字またはワイド文字を含む文字定数は、エラー検査のコンパイル・オプションにより警告の診断メッセージが出され、整数値として格納されます。2 文字以上の文字定数は、コモン C との互換性のために最終文字が最下位バイトで表現されます。基本実行文字集合には存在しない、8 進または 16 進のエスケープ・シーケンスを含む整数文字定数の表現は、エスケープ・シーケンスの 8 進数または 16 進数によって指定される値です。その値は、`unsigned` のコンパイル時オプションが有効かどうかに応じて符号付きまたは符号なし `char` として解釈されます。

`wchar_t` ワイド文字定数の型は `unsigned int` です。

B.13 文字列リテラル (§3.1.4)

同一の文字列リテラルは識別する必要がないと規格では規定されているため、文字列リテラルを変更した場合の動作は定義されていません。したがって、文字列リテラルまたはワイド文字列リテラルを変更すると、エラーになります。

B.14 演算子 — 複合代入 (§3.1.5)

旧形式の複合代入演算子 (`=+`, `-=`, `=*`, `=/`, `=%` など) は、規格には定義されていません¹。したがって、式 `=単項演算子 式` といった形式を持つ式では、`=` 単項演算子が以前は代入演算子と解釈されていましたが、`=` 単項演算子は現在では 2 つのトークン、すなわち代入演算子と単項演算子とに解釈されます。

エラー検査オプションを指定した場合は、`--`, `=*`, `=&`, `+=` (空白が入らない) に対して警告メッセージが出され、この意味の変更を知らせます。エラー検査オプションを指定しない場合には、メッセージは出されません。

B.15 文字と整数 — 値保存拡張 (§3.2.1.1)

初期バージョンの C では、汎整数拡張規則の実現に対して 2 つの異なる方法が取られていました。最初の方法は符号なし保存と呼ばれ、`unsigned char` および `unsigned short` が `unsigned int` へ拡張されます。2 番目の方法は値保存と呼ばれ、値を表現できる場合には `unsigned char` および `unsigned short` が `signed int` へ拡張されます。表現できない場合には `unsigned int` へ拡張されます。規格では、汎整数拡張は値保存であると規定されています。コモン C および VAX C モード以外のすべてのモードではこの方法に従っており、符号なし保存の算術変換に依存するプログラムは別の方法で解釈されます。

符号なし保存規則に依存する算術変換を検索できるようにするため、値保存規則の影響を受けるような `unsigned char` および `unsigned short` から `int` への汎整数拡張には、エラー検査オプションを使用するとフラグが付けられます。

¹ 初期バージョンの C では、複合代入演算子を定義された順 (`+=`, `-=`, `*=`) ではなく、逆順 (`=+`, `-=`, `=*`) で記述できた。この旧形式では、2 番目の演算子が有効な単項演算子でもあったため、複合代入演算子の構文が曖昧になった。

B.16 符号付き整数と符号なし整数の変換 (§3.2.1.2)

符号付き整数へ縮小された整数の値が大きすぎて表現できない場合、結果は切り捨てられ、余分の上位ビットは破棄されます。これは、コモン C および VAX C に互換性があります。

同じサイズの符号付き整数と符号なし整数の間の変換では、表現の変更はありません。

B.17 浮動小数点数と汎整数の変換 (§3.2.1.3)

整数を浮動小数点数に変換して正確に表現できない場合、変換の結果は正確に表現できる最も近い値になります。この結果はハードウェア上での変換の自然の結果であって、元の値より大きくなったり小さくなったりすることがあります。

コンパイル時に浮動小数点数を整数または別の浮動小数点型に変換して結果を表現できない場合、コンパイラは診断メッセージを出します。

汎整数または倍精度浮動小数点数を浮動小数点数に変換して元の値を正確に表現できない場合、結果は float 型の最も近い値に丸められます。詳細については、『*MIPS R-Series Processor Architecture Manual*』または『*VAX Architecture Manual*』など使用しているプラットフォームに固有のアーキテクチャ・マニュアルを参照してください。

double 値を float に縮小する際に、変換する値が表現できる値の範囲にあるのに正確に表現できない場合、結果は最も近い上位値または下位値になります。Compaq C では、結果を最も近い表現可能な float 値に丸めます。

同様の丸めは、long double から double または float へ縮小する際にも行われます。

B.18 ポインタ変換 (§3.2.2.3)

たとえ 2 つの型が同じ表現であっても (int と long のように), 型はやはり異なります。つまり, int へのポインタを long へのポインタに割り当てるには, キャスト演算子を使用しなければなりません。

この規則は, コモン C および VAX C モードでは緩和されます。ポインタ変換では表現の変更はありませんが, 一部のマシンでは境界調整制約のため未調整のポインタによるアクセスは, アクセス時間の遅延, マシン例外, または予想できない結果を生じることがあります。

B.19 構造体および共用体メンバ (§3.3.2.3)

値を保持しているメンバとは異なる共用体メンバにアクセスした場合の結果は, メンバのデータ型とその共用体内での境界調整によって異なります。

B.20 sizeof 演算子 (§3.3.3.4)

sizeof 演算子の型は size_t です。この型は配列の最大サイズを保持するために必要な整数の型ですが, *Compaq C* はこの型を <stddef.h> ヘッダの中で unsigned int として定義しています。

B.21 キャスト演算子 (§3.3.4)

ポインタは汎整数型に変換できると規格では規定していますが, 必要な整数のサイズと結果は処理系定義です。ポインタは, int 型または long 型 (あるいはその unsigned の同値) のオブジェクトと同じ記憶量を占有します。したがって, ポインタはこれらの整数型のいずれにも変換することができ, 値を変更せずに元に戻すことができます。スケーリングは行われず, 値の表現は変化しません。

ポインタと `char` などの短整数型の間の変換は、`unsigned long` 型のオブジェクトと短整数型の間の変換と同様です。ポインタの上位ビットは破棄されます。短整数とポインタの間の変換は、短整数型と `unsigned long` の間の変換と同様です。短整数型が符号付きの場合、ポインタの上位ビットは符号ビットのコピーで充填されます。エラー検査のコンパイル・オプションを使用した場合には、これらの型のキャスト演算に対してメッセージが出されます。

B.22 乗除演算子 (§3.3.5)

規格では、除算演算子および剰余演算子に移植可能な意味を与えていません。

Compaq C では次の意味に従います。

- 除算演算子 (`/`) のどちらか一方のオペランドが負の場合、結果は 0 に近い値 (代数商より小さい最も大きな整数) に切り捨てられます。
- 剰余演算子 (`%`) のどちらか一方のオペランドが負の場合、結果の符号は最初のオペランドの符号と同じです (コモン C、MIPS C、および VAX C との互換性のため)。

次のような未定義動作がコンパイル時に検出された場合、コンパイラは警告を出します。

- 整数オーバーフロー
- 0 による除算
- 0 による剰余

B.23 加減演算子 (§3.3.6)

同じ配列のメンバへのポインタで減算を行うことができます。結果は 2 つの配列メンバの間の要素の数であり、結果の型は `ptrdiff_t` です。*Compaq C* では、この型を `int` と定義しています。

B.24 ビット単位のシフト演算子 (§3.3.7)

`E1 >> E2`の結果は、`E2`ビット位置だけ右へシフトした `E1` です。`E1` が符号付き型の場合、結果の値は空いた上位ビットに `E1` の符号ビットのコピーが充填された `E1` のシフト値です (算術シフト)。

B.25 記憶域クラス指定子 (§3.5.1)

`register`記憶域クラス指定子は、オブジェクトへのアクセスをできるだけ速く行うように指定します。`register` の指定は、変数をレジスタに格納する確率を高くするために行われます。ただし、コンパイラのレジスタ割当て技法により、`register` キーワードはもはや使用されなくなりました。つまり、*Compaq C*ではすべての `register` 要求を受け入れて無視します。

B.26 型指定子 (§3.5.2)

コモン C および VAX C との互換性のため、`long float` の組み合わせは `double` の同義語としてサポートされます。ただし、省略時のモードまたは厳密な ANSI モードでコンパイルした場合にこの組み合わせを使用すると警告が出されます。

B.27 構造体および共用体指定子 (§3.5.2.1)

`int`ビット・フィールドの最上位ビット位置は、VAX C 互換性モード以外では符号ビットとして扱われません。つまり、`int`型はどのビット・フィールド型の場合も、`unsigned int` と同じ型であることを指定します。VAX C モードでは、`int`型はどのビット・フィールド型の場合も、`signed int` と同じ型であることを指定します。

B.28 変形構造体および共用体

変形構造体および共用体はVAX Cの拡張機能であり，ネストした構造体および共用体を外側の集合体のメンバとして宣言できます。これにより，これらのメンバの参照時に中間修飾子を指定する必要がなくなります。これらの機能は，VAX Cモードでのみ利用可能です。

これらの拡張機能についての詳細は，プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

B.29 構造体の境界調整

構造体の境界調整とサイズは，各プラットフォームにおける構造体の構成要素の境界調整条件とサイズの影響を受けます。構造体はどのバイト境界からでも開始することができ，整数バイト数であれば任意のバイト数を占有できます。ただし，個々のアーキテクチャまたはオペレーティング・システムは，特定の境界調整およびパディング条件（プラグマおよびコマンド行オプションで変更可能）を指定することがあります。

OpenVMS Alpha と Tru64 UNIX の場合

OpenVMS Alpha および *Tru64 UNIX* システムでは，非ビット・フィールドの構造体メンバは，省略時の設定では自然に境界調整されます。

省略時の構造体の境界調整は，構造体内の任意のメンバが必要とする最大の境界調整です。構造体はサイズ（バイト単位）が境界調整条件の倍数になるようにパッドされて，構造体または共用体が配列のメンバである場合に適切な境界調整が行えるようにします。

構造体の構成要素は，宣言された順にメモリ内に格納されます。最初の構成要素は構造体全体と同じアドレスです。構成要素間にパディングを挿入して，個々の構成要素の境界調整条件を満たします。

ビット・フィールドはどんな汎整数型でも構いません。ただし，型が `int`，`unsigned int`，または `signed int` 以外の場合に，コンパイラはエラー検査オプション指定時には警告を出します。ビット・フィールドが存在すると，構造体または共用体全体

の境界調整は少なくともビット・フィールドの基本型の境界調整と同じになります。

他のビット・フィールドの直後に宣言されないビット・フィールド (長さ 0 のビット・フィールドを含む) では、境界調整条件はその基本型が課したものとなります。ビット・フィールドは、(ビット・フィールドの基本型と同じサイズの) 境界調整単位内で下位から上位へ割り当てられます。

#pragma member_alignmentが有効に設定されていると、あるビット・フィールドが別のビット・フィールドの直後に続くとき、十分な領域が残っている場合にはそのビットは同じ単位内の隣接する領域へバックされます。十分な領域が残っていない場合には最初のビット・フィールドの終わりにパディングが挿入され、2 番目のビット・フィールドは次の単位へ置かれます。

#pragma nomember_alignmentが有効に設定されていると、ビット・フィールドは記憶域単位境界にまたがることができます。Alpha システムでは省略時にはmember_alignmentに設定され、VAX システムでは省略時にはnomember_alignmentに設定されます。

char基本型のビット・フィールドは、8 ビット以上でなければなりません。short基本型のビット・フィールドは、16 ビット以上でなければなりません。

OpenVMS VAX の場合

OpenVMS VAXシステムでは、構造体または構造体メンバを特定の境界に調整する必要はなく、非ビット・フィールド構造体メンバは特に指定しない限りバイトで調整されます。

構造体の構成要素は、宣言された順にメモリ内に格納されます。最初の構成要素は構造体全体と同じアドレスです。各構成要素は、それぞれ前の構成要素の直後に次々に格納されます。

次のプラグマを使用すると、構造体メンバを通常の境界調整にすることができます。

```
#pragma member_alignment
```

『*Compaq C for OpenVMS システム ユーザーズ・ガイド*』には、*OpenVMS VAX*上における構造体の境界調整の例と図が掲載されています。

ビット・フィールドはどの汎整数型でも構いません。ただし、`/STANDARD=ANSI89`が指定され、型が `int`、`unsigned int`、または `signed int` 以外の場合にはコンパイラは警告を出します。ビット・フィールドは、単位内で下位から上位へ割り当てられます。あるビット・フィールドが別のビット・フィールドの直後に続く場合は、たとえそれが別のバイトへオーバーフローしてもそのビットは隣接する領域へバックされます。ただし、名前のないビット・フィールドを長さ 0 と指定した場合には、直後のビット・フィールドが次のバイト境界から始まるようにパディングが挿入されます。

『*Compaq C for OpenVMS システム ユーザーズ・ガイド*』に、*OpenVMS VAX*上におけるビット・フィールドの境界調整の例と図が掲載されています。

B.30 列挙型指定子 (§3.5.2.2)

規格では、列挙型はそれぞれ処理系定義の整数型と互換性があることを規定しています。*Compaq C*では、列挙型はそれぞれ `signed int`型と互換性があります。

B.31 型修飾子 (§3.5.3)

`volatile`記憶域クラスは、コンパイラに未知の方法で変更できる変数に対して指定されます。したがって、あるオブジェクトが `volatile` と宣言された場合、ソース・コードでこのオブジェクトを参照するたびにオブジェクト・コードでメモリを参照することになります。

B.32 宣言子 (§3.5.4)

算術型、構造体型、共用体型、または不完全型を変更できるポインタ、関数、または配列宣言子の数に内部限界はありません。

B.33 初期化 (§3.5.7)

C では、論理的に必要なではない場合にも初期化子の中括弧 { } で囲むことができます。これにより、部分的に無視される中括弧を持つ集合体の初期化子は、実現されたパーサの型に応じて異なる方法で解析されます (ボトムアップまたはトップダウン)。規格では、Kernighan と Ritchie の『The C Programming Language, 2nd Edition』の中で初めて指定されたトップダウンを指定しています。部分的に中括弧で囲まれた初期化子がボトムアップ解析 (コモン C 解析) に依存するプログラムは、予想外の結果を生じることがあります。この構成要素は許されていますが、コモン C モード、またはエラー検査オプションを使用した場合には警告メッセージが出され、中括弧が無視されることを通知します。

B.34 switch 文 (§3.6.4.2)

switch 文中の case ラベルの数に制限はありません。

B.35 外部オブジェクト定義 (§3.7.2)

コモン C モードでは、すべての extern オブジェクトはファイル・スコープを持ちます。

B.36 条件付き取込み (§3.8.1)

以前のプリプロセッサでは、次の例のように前処理命令の後に余分なテキストを記述することが許されていました。

```
#endif system1
```

しかし規格では、前処理命令の後に許されるテキストはコメントだけであることが規定されています。したがって、この構文規則に違反した場合、*Compaq C* コンパイラは警告メッセージを出します。

`#if` および `#elif` 命令内の文字定数の数値は、これらの命令の一部ではない式の中で同一の文字定数が使用されたときに得られた値と一致します。

B.37 ソース・ファイルの取込み (§3.8.2)

二重引用符で囲んだパス名 (`#include "stdio.h"`) または山括弧で囲んだパス名 (`#include <stdio.h>`) を使用して、ソース・ファイルを取り込むことができます。また、*OpenVMS* システムではテキスト・ライブラリからモジュールを取り込む方法もサポートします。ソース・ファイルを取り込むための検索パス・アルゴリズムについては、プラットフォームに固有の *Compaq C* ドキュメントを参照してください。

B.38 マクロ置換 — あらかじめ定義されたマクロ名 (§3.8.3)

規格であらかじめ定義されたマクロ名の他に、*Compaq C* コンパイラではさまざまなシステム識別用に別の前処理マクロを定義しています。コンパイラが起動されると、オペレーティング・システム、アーキテクチャ、言語、コンパイラ・モード、および他の環境変数に従って、適切な識別マクロが定義されます。これらのマクロは `#ifdef` 前処理命令で参照すると、特定の環境に適用するコードを分離することができます。

Compaq C の各プラットフォームにはあらかじめ定義された追加のマクロがあるかもしれません。詳細については、プラットフォームに固有の *Compaq C* ドキュメントを参照してください。

表 B-1 に掲載している形式はすべて、表示されたプラットフォーム上で定義されています。ただし、厳密な ANSI モードが有効な場合には新しいつづりだけが定義されます。

表 B-1 に示しているのは、*Tru64 UNIX* 上であらかじめ定義されたマクロ名です。

表 B-1 Tru64 UNIX 上であらかじめ定義されたマクロ名

	マクロ名
オペレーティング・システム名:	unix __unix__ __osf SYSTYPE_BSD _SYSTYPE_BSD
アーキテクチャ名:	__alpha
製品名:	__DEC C __DEC C_VER LANGUAGE_C __LANGUAGE_C__

表 B-2 に示しているのは、*OpenVMS VAX* および *Alpha* システム上であらかじめ定義されたマクロ名です。厳密な ANSI モードが有効になっていない場合に限りすべての形式が定義されます。厳密な ANSI モードが有効になっている場合には新しいつづりだけが定義されます。

表 B-2 OpenVMS VAX および Alpha 上であらかじめ定義されたマクロ名

	新しいつづり	従来につづり
オペレーティング・システム名:	__vms __VMS __vms_version __VMS_VERSION	vms VMS vms_version VMS_VERSION
アーキテクチャ名:	__vax (<i>VAX</i>) __VAX (<i>VAX</i>) __alpha (<i>Alpha</i>) __ALPHA (<i>Alpha</i>) __Alpha_AXP (<i>Alpha</i>)	vax (<i>VAX</i>) VAX (<i>VAX</i>) — — —

(次ページに続く)

表 B-2 (続き) OpenVMS VAX および Alpha 上であらかじめ定義されたマクロ名

	新しいつづり	従来のつづり
	<code>__32BITS</code> (<i>Alpha</i>)	—
製品名:	<code>__vaxc</code>	<code>vaxc</code>
	<code>__VAX C</code>	<code>VAX C</code>
	<code>__vax11c</code>	<code>vax11c</code>
	<code>__VAX11C</code>	<code>VAX11C</code>
	<code>__STDC__</code>	—
	<code>__DEC C</code>	—
	<code>__DEC C_VER</code>	—
	<code>__VMS_V6_RTL_COMPAT</code>	—
コンパイラ・モード:	<code>__DEC C_MODE_STRICT</code>	—
	<code>__DEC C_MODE_RELAXED</code>	—
	<code>__DEC C_MODE_VAX C</code>	—
	<code>__DEC C_MODE_COMMON</code>	—
浮動小数点:	<code>__D_FLOAT</code>	—
	<code>__G_FLOAT</code>	—
	<code>__IEEE_FLOAT</code> (<i>Alpha</i>)	—
	<code>__X_FLOAT</code> (<i>Alpha</i>)	—
その他:	<code>__HIDE_FORBIDDEN_NAMES</code>	—
	<code>__INITIAL_POINTER_SIZE</code> (<i>Alpha</i>)	—

表 B-3 に示すマクロを明示的に定義することにより，どの C ライブラリ・ルーチンをヘッダ・ファイルで宣言するかを制御したり，規格準拠チェックを行うことができます。これらのマクロを定義するには，次のいずれかを使用します。

- `-D` フラグ (*Tru64 UNIX*)
- `/DEFINE` 修飾子 (*OpenVMS*)

- #define前処理命令

表 B-3 ライブラリ・ルーチン規格準拠マクロ — 全プラットフォーム

マクロ	規格
_XOPEN_SOURCE_EXTENDED	XPG4-UNIX
_XOPEN_SOURCE	XPG4
_POSIX_C_SOURCE	POSIX
_ANSI_C_SOURCE	ISO C および ANSI C
_AES_SOURCE (<i>Tru64 UNIX</i>)	アプリケーション環境サービス
_OSF_SOURCE (<i>Tru64 UNIX</i>)	OSF 互換性
_VMS_V6_SOURCE (<i>OpenVMS</i>)	<i>OpenVMS</i> バージョン 6 互換性
_DEC C_V4_SOURCE (<i>OpenVMS</i>)	<i>DEC C</i> バージョン 4 互換性

B.39 ##演算子 (§3.8.3.3)

マクロ置換並び内に ##演算子があると、演算子の両側の 2 つのトークンが連結されて、単一のトークンになります。

コモン C および VAX C の互換性モードでは、コメントはマクロ呼出し後に空文字列に置き換えられるため、コメントも 2 つのトークンを連結することができます。

厳密な ANSI モードまたは省略時のモードでは、コメントは単一の空白に置き換えられるので、この動作はサポートされません。

B.40 エラー命令 (§3.8.5)

#error命令はエラー・メッセージを出して、コンパイルを終了します。

B.41 プラグマ命令 (§3.8.6)

規格が承認している言語に拡張機能を追加する方式は、プラグマを追加して行われます。認識されないプラグマには情報メッセージで診断されます。サポートされるプラグマはプラットフォームによって異なります。詳細については、プラットフォームに固有の *Compaq C* ドキュメントを参照してください。

ファイルの前処理だけを行う場合には、*Compaq C* で認識されるすべてのプラグマは変更なしで出力へ書き込まれます。

B.42 関数のインライン展開

関数のインライン展開は、プロシージャ呼出しのオーバーヘッドを排除して、一般的な最適化方式を展開されたコードに適用することができます。関数インラインがマクロに比べて優れている点は、実引数が 1 回だけ評価され、優先順位の問題を避けるために括弧を多用する必要がなく、また、実際の展開をコマンド行から制御できることです。

次のプラグマを使用すると、関数のインライン展開を制御することができます。

```
#pragma inline (関数名 [, 関数名....])  
#pragma noline (関数名 [, 関数名....])
```

関数が `inline` 命令の中で指定されたとき、その関数に次の特性がある場合には、この関数への呼出しはインライン・コードとして展開されます。

- 関数が `noline` 命令の中で指定された場合、その関数への呼出しはインライン・コードとして展開されません。
- 関数が `inline` または `noline` 命令の中で指定されなかった場合、コンパイラは経験的手法により適切な場合には呼出しのインライン展開を実行します。
- 関数が `inline` および `noline` 命令の両方で指定された場合、コンパイラはエラーを出します。

`noline` のコンパイル・オプションを使用した場合、このオプションはすべての `inline` プラグマ命令を無効にします。

インライン関数の特性は次のとおりです。

- インライン関数は再帰的に使用できますが、その場合には、1 レベルのインライン展開だけが実行されます。
- インラインされる関数の定義を含んでいるソース・ファイルの中での呼出しだけが、インライン展開されます。
- インライン関数のアドレスを取得することができ、インラインされた関数名をアドレスへ変換する式が許されます。
- `varargs` パッケージ (関数が可変個の実引数を取ることを許す) の使用は、インライン関数では許されません。
- インライン関数は、実引数並びで省略記号を使用して宣言することはできません。

B.43 結合プラグマ

Compaq C は、*OpenVMS Alpha* システム上において `#pragma linkage` および `#pragma use_linkage` 前処理命令をサポートします。

これらのプラグマは特殊な結合特性を定義する場合、およびこれらの結合特性を関数と関連付ける場合に使用します。詳細については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

B.44 その他のプラグマ

次のプラグマは、VAX C 互換性モードだけで提供されます。

```
#pragma dictionary CDD_path  
#pragma module title ident
```

これらのプラグマは、`#dictionary` 命令と `#module` 命令にそれぞれ対応します。

システムでサポートされている追加のプラグマについては、プラットフォーム固有の *Compaq C* マニュアルを参照してください。

C

ASCII 等価テーブル

表 C-1 に、ASCII 文字集合を示します。各文字の 8 進値、10 進値、および 16 進値を示しています。

表 C-1 ASCII 等価チャート

文字	8 進値	10 進値	16 進値
\0	00	0	0x0
\001	01	1	0x1
\002	02	2	0x2
\003	03	3	0x3
\004	04	4	0x4
\005	05	5	0x5
\006	06	6	0x6
\007	07	7	0x7
\b	010	8	0x8
\t	011	9	0x9
\n	012	10	0xA
\v	013	11	0xB
\f	014	12	0xC
\r	015	13	0xD
\016	016	14	0xE
\017	017	15	0xF
\020	020	16	0x10
\021	021	17	0x11

(次ページに続く)

ASCII 等価テーブル

表 C-1 (続き) ASCII 等価チャート

文字	8 進値	10 進値	16 進値
\022	022	18	0x12
\023	023	19	0x13
\024	024	20	0x14
\025	025	21	0x15
\026	026	22	0x16
\027	027	23	0x17
\030	030	24	0x18
\031	031	25	0x19
\032	032	26	0x1A
\033	033	27	0x1B
\034	034	28	0x1C
\035	035	29	0x1D
\036	036	30	0x1E
\037	037	31	0x1F
(空白)	040	32	0x20
!	041	33	0x21
"	042	34	0x22
#	043	35	0x23
\$	044	36	0x24
%	045	37	0x25
&	046	38	0x26
\	047	39	0x27
(050	40	0x28
)	051	41	0x29
*	052	42	0x2A
+	053	43	0x2B
,	054	44	0x2C
-	055	45	0x2D

(次ページに続く)

表 C-1 (続き) ASCII 等価チャート

文字	8 進値	10 進値	16 進値
.	056	46	0x2E
/	057	47	0x2F
0	060	48	0x30
1	061	49	0x31
2	062	50	0x32
3	063	51	0x33
4	064	52	0x34
5	065	53	0x35
6	066	54	0x36
7	067	55	0x37
8	070	56	0x38
9	071	57	0x39
	072	58	0x3A
;	073	59	0x3B
<	074	60	0x3C
=	075	61	0x3D
>	076	62	0x3E
?	077	63	0x3F
—	0010	64	0x40
A	0101	65	0x41
B	0102	66	0x42
C	0103	67	0x43
D	0104	68	0x44
E	0105	69	0x45
F	0106	70	0x46
G	0107	71	0x47
H	0110	72	0x48
I	0111	73	0x49

(次ページに続く)

ASCII 等価テーブル

表 C-1 (続き) ASCII 等価チャート

文字	8 進値	10 進値	16 進値
J	0112	74	0x4A
K	0113	75	0x4B
L	0114	76	0x4C
M	0115	77	0x4D
N	0116	78	0x4E
O	0117	79	0x4F
P	0120	80	0x50
Q	0121	81	0x51
R	0122	82	0x52
S	0123	83	0x53
T	0124	84	0x54
U	0125	85	0x55
V	0126	86	0x56
W	0127	87	0x57
X	0128	88	0x58
Y	0130	89	0x59
Z	0131	90	0x5A
[0133	91	0x5B
\	0134	92	0x5C
]	0135	93	0x5D
^	0136	94	0x5E
_	0137	95	0x5F
`	0140	96	0x60
a	0141	97	0x61
b	0142	98	0x62
c	0143	99	0x63
d	0144	100	0x64
e	0145	101	0x65

(次ページに続く)

表 C-1 (続き) ASCII 等価チャート

文字	8 進値	10 進値	16 進値
f	0146	102	0x66
g	0147	103	0x67
h	0150	104	0x68
i	0151	105	0x69
j	0152	106	0x6A
k	0153	107	0x6B
l	0154	108	0x6C
m	0155	109	0x6D
n	0156	110	0x6E
o	0157	111	0x6F
p	0160	112	0x70
q	0161	113	0x71
r	0162	114	0x72
s	0163	115	0x73
t	0164	116	0x74
u	0165	117	0x75
v	0166	118	0x76
w	0167	119	0x77
x	0170	120	0x78
y	0171	121	0x79
z	0172	122	0x7A
{	0173	123	0x7B
	0174	124	0x7C
}	0175	125	0x7D
~	0176	126	0x7E
\177	0177	127	0x7F

Compaq C でサポートするコモン C の拡張

*Compaq C*では、ANSI C 規格に対するいくつかのコモン C (旧形式の C) の機能拡張をサポートします。これらの拡張は、コンパイル・コマンド行でコモン C 互換性オプションを使用した場合にだけ認識されます。コモン C の拡張を使用すると、c89 コンパイラを使用して、当初はポータブル C コンパイラ (pcc) 用に記述されたコードをコンパイルすることができます。

以降の各節で、コモン C 互換性オプションで利用可能なコモン C の拡張機能について説明します。ANSI C 規格に対する拡張は、次の 2 つのカテゴリに分けられます。

- コモン C 互換性オプションを指定せずにコンパイルすると、診断メッセージを引き起こす ANSI C プログラムと互換性のある拡張。
- ANSI C プログラムと互換性がなく、コモン C 互換性オプションを指定せずに使用すると異なるコンパイラ動作を引き起こすおそれのある拡張。

D.1 ANSI C と互換性のある拡張

- 緩やかなポインタとポインタまたはポインタと整数の互換性が許されます。つまり、すべてのポインタ型と整数型は互換性があり、ポインタ型は、示すオブジェクトの型に関係なく、相互に互換性があります。したがって、コモン C オプションの指定時には、float へのポインタは、int へのポインタと互換性があります。
- 数字 8 と 9 が 8 進整数定数で有効です (ただし、コンパイラが警告メッセージを出す)。

- ビット・フィールド・データ型には `enum` , `short` , `char` , および `long` があります。ANSI C 規格では `int` , `unsigned int` , または `signed int` だけが許されません。
- `long float` は `double` の同義語と認識されます。
- `main()`関数の 3 番目の実引数、すなわち `char *envp[]` が許されます¹。

Compaq C をコモン C 互換性モードで実行する場合、`main`関数は 3 番目の仮引数である `envp`環境配列を受け入れることができます。この配列には、ユーザ名などのプロセス情報と制御情報が含まれ、コマンド行から渡す実引数とは無関係です。主に `exec` および `getenv`ライブラリ関数の呼出し時に使用されます。

使用しているホスト環境で `main`関数を呼び出すための詳細については、プラットフォームに固有の *Compaq C* のマニュアルを参照してください。

- `#else` および `#endif`前処理命令の後にテキストが許されます。
- アドレス定数を `int` にキャストすることができます。
- コンパイルの完了時に存在する仮定義は、従来の定義解決モデルに従ってリンカへの仮定義になります。
- 表現の変更を引き起こさないキャストは、左辺値として有効です。
- 暗黙の関数宣言はブロック・レベルではなく、ファイル・レベルで作成されます。
- `int`型と `long`型は互換性があります。
- `register`記憶域クラスの指定された変数のアドレスを取得することが許されません。
- `static`記憶域クラスでの関数のブロック・レベルの宣言が許されます。
- 配列型では、要素の型は不完全でも構いません。
- 仮定義された変数の型は、コンパイル単位の終了時に不完全でも構いません。この場合には警告が出されます。
- `case` ラベルの値にポインタ型が許されます。
- 列挙並びで後続 (余分) のコンマが許されます。

¹ `main()`関数の仮引数は、厳密な ANSI モードでのみ検査される。

- 最後の構造体または共用体メンバの後のセミコロンは省略できます。
- キャリッジ・リターンは受け入れられ、空白として扱われます。

D.2 ANSI C と互換性のない拡張

- 符号なし保存規則が適用されます。つまり、unsigned char および unsigned short は unsigned int へ拡張します。
- コメントは単一の空白ではなく、スペースなしへ変換されて、トークン連結が可能になります (コンパイラは 2 つの隣接するトークンを連結しようとする)。
- extern オブジェクトはすべて、ファイル・スコープを持ちます。
- マクロ定義の文字列または文字定数内では、マクロ仮引数が認識されて置き換えられます。
- マクロ置換の際、実引数の前処理トークンのマクロ置換が行なわれるのは、マクロが展開される前ではありません。
置換並びの再スキャンの際に、置き換えられるマクロの名前が見つかった場合、マクロ置換が行なわれます。
- ANSI C 規格に準拠しない、あらかじめ定義されたマクロ名のサポート。つまり、マクロ名を指定する際に、先頭の大文字の前に 1 つまたは複数のアンダスコアをつけていないものです。
- 前処理命令は、行の先頭のコラムに # 開始文字が現れた場合にだけ認識されます。空白が先行する前処理命令は無視されます。
- #ifdef は “#if defined” として扱われます。
- #ifndef は “#if !defined” として扱われます。
- マクロ置換並び中のコメントは、連結後に有効なトークンが生成される場合には、隣接する空白が削除されない点を除き、## 演算子のように動作します。生成されたトークンが有効ではない場合には、マクロ置換並び中のコメントが削除されます。
- 3 文字表記は認識も置換もされません。

Compaq C でサポートする VAX C の拡張

*Compaq C*では、ANSI 規格の C に対するいくつかの VAX C の機能拡張をサポートします。これらの拡張は、コンパイル・コマンド行で VAX C 互換性オプションを使用した場合にだけ認識されます。VAX C の拡張機能により、*Compaq C* コンパイラを使用して、当初は VAX C コンパイラ用に記述されたコードをコンパイルすることができます。

以降の各節で、VAX C 互換性オプション指定時に利用可能な VAX C の拡張について説明します。ANSI 規格の C 言語に対する拡張は、次の 2 つのカテゴリに分けられます。

- VAX C 互換性オプションを指定せずにコンパイルすると、診断メッセージを引き起こす ANSI C プログラムと互換性のある拡張。
- ANSI C プログラムと互換性がなく、VAX C 互換性オプションを指定せずに使用すると異なるコンパイラ動作を引き起こすおそれのある拡張。

E.1 ANSI C と互換性のある拡張

- VAX C 固有のプラグマは認識されます。
- 緩やかなポインタとポインタまたはポインタと整数の互換性が許されます。つまり、すべてのポインタ型と整数型は互換性があり、ポインタ型は示すオブジェクトの型に関係なく、相互に互換性があります。したがって、VAX C オプションの指定時には、float へのポインタは、int へのポインタと互換性があります。
- #module 命令が許されます。*Tru64 UNIX* システムでは、この命令は警告メッセージを出して無視されます。

- #dictionary命令が許されます。Tru64 UNIXシステムでは、この命令は警告メッセージを出して無視されます。
- モジュール形式の #include が許されます。Tru64 UNIXシステムでは、この命令のモジュール形式はエラーを生じます。
- VAX Cモードでは、ビット・フィールドの型に int を指定することは、signed int を指定することと同等です。
- 組み込み関数が認識されます。
- main_programオプションを使用して、特定の関数を所定のプログラムの main 関数として識別することができます。

VAX Cモードでコンパイルする場合、プログラム内で main 関数を指定するもうひとつの方法として、次のオプションを関数定義に組み込むことができます。

main_program

このオプションはキーワードではなく、大文字でも小文字でも記述することができます。main_programオプションは、メイン・プログラムに main 以外の名前を許す場合に役立ちます。

プロトタイプ形式の関数定義では、次の例に示すように関数宣言部と左中括弧の間に main_program を組み込みます。

```
char lower(int c_up)
main_program
{
    .
    .
    .
}
```

旧形式の関数定義では、次の例に示すように、プロトタイプ形式と同じ場所ですが、仮引数宣言の前に main_program を組み込みます。

```
char lower(c_up)
main_program
int c_up;
{
    .
    .
    .
}
```

どちらの例も、lower関数を main 関数として設定します。実行は、関数のリンク順に関係なく、そこから始まります。

- ビット・フィールド・データ型には enum, short, char, および long があります。ANSI C 規格では int, unsigned int, または signed int だけが許されます。
- 構造体の最後のメンバは、サイズを指定しない配列でも構いません。
- 2 つの struct 型または 2 つの union 型は、サイズが同じ場合には同じ型とみなされます。
- static 記憶域クラスでの関数のブロック・レベルの宣言が許されます。
- 定数のアドレスを関数に引き渡すことができます。
- register 記憶域クラスの指定された変数のアドレスを取得することが許されます。
- main() 関数の 3 番目の実引数、すなわち char *envp[] が許されます。

Compaq C を VAX C 互換性モードで実行する場合、main 関数は 3 番目の仮引数である envp 環境配列を受け入れることができます。この配列には、ユーザ名などのプロセス情報と制御情報が含まれ、コマンド行から渡す実引数とは無関係です。主に、exec および getenv ライブラリ関数の呼出し時に使用されます¹。

使用しているホスト環境で main 関数を呼び出すための詳細については、プラットフォームに固有の Compaq C のマニュアルを参照してください。

- long float は double の同義語と認識されます。
- 複数の文字を含む文字定数は、little endian 順にパックされます。たとえば、'AB' は 'A'<<8+'B' ではなく 'B'<<8+'A' として扱われます。

¹ main() 関数の仮引数は、厳密な ANSI モードでのみ検査される。

- 列挙並びで、後続 (余分) のコンマが許されます。
- 配列の要素の型は不完全でも構いません。
- キャリッジ・リターンは受け入れられ、空白として扱われます。

E.2 ANSI C と互換性のない拡張

- 符号なし保存規則が適用されます。つまり、unsigned char および unsigned short は unsigned int へ拡張します。
- VAX C固有のあらかじめ定義されたマクロが認識されます。
- VAX C固有のキーワードが認識されます。
- マクロ仮引数は、マクロ定義中の文字列または文字定数として認識されて置き換えられます。
- コメントは単一の空白ではなく、空白なしへ変換されて、トークン連結が可能になります (コンパイラは2つの隣接するトークンを連結しようとします)。
- マクロ置換並び中のコメントは、連結後に有効なトークンが生成される場合には、隣接する空白が削除されない点を除き ##演算子のように動作します。生成されたトークンが有効ではない場合には、マクロ置換並び中のコメントは削除されます。
- 3文字表記は認識も置換もされません。
- 変形構造体および共用体が許されます。

変形構造体および共用体宣言を使用すると、中間構造体または共用体識別子を参照しなくても、ネストした集合体のメンバを参照することができます。変形構造体または共用体宣言を別の構造体または共用体宣言の中にネストすると、ネストされた変形集合体は別の集合体として存在せず、*Compaq C*はそのメンバを外側の集合体にコピーします。

変形構造体および共用体の宣言は、`variant_struct` および `variant_union` キーワードを使用して行います。これらの宣言の形式は、次の点を除いて通常の構造体または共用体の形式と同じです。

- 変形集合体は、他の有効な構造体または共用体宣言の中にネストしなければなりません。
- 変形集合体宣言ではタグを使用することはできません。
- 変形集合体宣言では少なくとも 1 つのメンバを宣言しなければならず、そのメンバはポインタや配列として宣言することはできません。

変形構造体または共用体の初期化は、通常の構造体または共用体の初期化と同じです。

VAX C 互換性オプションを使用した場合、代入演算における 2 つの構造体または共用体はサイズが同じであればよく、メンバおよびメンバ型が同じである必要はありません。

次の例は、変形構造体宣言の形式と変形構造体のメンバへの参照方法を示しています。

```
#include <stdio.h>
enum packet_type {TEXT, INTEGER};

/* This structure can contain either a text_packet or an integer value.
   It can only contain one of these at a time, since they share the same
   storage. */

struct packet
{
    enum packet_type type;
    variant_union
    {
        variant_struct
        {
            int str_size;
            char *text;
        } text_packet;
        variant_struct
        {
            int value;
        } value_packet;
    }
}
```

Compaq C でサポートする VAX C の拡張
E.2 ANSI C と互換性のない拡張

```
        } text_or_int;
    } packet = {TEXT, 24, "I love the color purple"};

main()
{
    if (packet.type == TEXT)
        printf(" %s. \n", packet.text);
    else
        printf(" %d \n", packet.value);

    packet.type = INTEGER;
    packet.value = 42;

    printf(" The meaning of life, the universe, and everything is: %d. \n",
packet.value);
}
```

F

識別子のユニバーサル・キャラクタ名

識別子のユニバーサル・キャラクタ名として使用できる 16 進コード値の一覧を次に示します。基本文字セットの範囲に含まれる部分は示していません。

Latin:	00AA, 00BA, 00C0-00D6, 00D8-00F6, 00F8-01F5, 01FA-0217, 0250-02A8, 1E00-1E9B, 1EA0-1EF9, 207F
Greek:	0386, 0388-038A, 038C, 038E-03A1, 03A3-03CE, 03D0-03D6, 03DA, 03DC, 03DE, 03E0, 03E2-03F3, 1F00-1F15, 1F18-1F1D, 1F20-1F45, 1F48-1F4D, 1F50-1F57, 1F59, 1F5B, 1F5D, 1F5F-1F7D, 1F80-1FB4, 1FB6-1FBC, 1FC2-1FC4, 1FC6-1FCC, 1FD0-1FD3, 1FD6-1FDB, 1FE0-1FEC, 1FF2-1FF4, 1FF6-1FFC
Cyrillic:	0401-040C, 040E-044F, 0451-045C, 045E-0481, 0490-04C4, 04C7-04C8, 04CB-04CC, 04D0-04EB, 04EE-04F5, 04F8-04F9
Armenian:	0531-0556, 0561-0587
Hebrew:	05B0-05B9, 05BB-05BD, 05BF, 05C1-05C2, 05D0-05EA, 05F0-05F2
Arabic:	0621-063A, 0640-0652, 0670-06B7, 06BA-06BE, 06C0-06CE, 06D0-06DC, 06E5-06E8, 06EA-06ED
Devanagari:	0901-0903, 0905-0939, 093E-094D, 0950-0952, 0958-0963
Bengali:	0981-0983, 0985-098C, 098F-0990, 0993-09A8, 09AA-09B0, 09B2, 09B6-09B9, 09BE-09C4, 09C7-09C8, 09CB-09CD, 09DC-09DD, 09DF-09E3, 09F0-09F1

識別子のユニバーサル・キャラクタ名

Gurmukhi:	0A02, 0A05-0A0A, 0A0F-0A10, 0A13-0A28, 0A2A-0A30, 0A32-0A33, 0A35-0A36, 0A38-0A39, 0A3E-0A42, 0A47-0A48, 0A4B-0A4D, 0A59-0A5C, 0A5E, 0A74
Gujarati:	0A81-0A83, 0A85-0A8B, 0A8D, 0A8F-0A91, 0A93-0AA8, 0AAA-0AB0, 0AB2-0AB3, 0AB5-0AB9, 0ABD-0AC5, 0AC7-0AC9, 0ACB-0ACD, 0AD0, 0AE0
Oriya:	0B01-0B03, 0B05-0B0C, 0B0F-0B10, 0B13-0B28, 0B2A-0B30, 0B32-0B33, 0B36-0B39, 0B3E-0B43, 0B47-0B48, 0B4B-0B4D, 0B5C-0B5D, 0B5F-0B61
Tamil:	0B82-0B83, 0B85-0B8A, 0B8E-0B90, 0B92-0B95, 0B99-0B9A, 0B9C, 0B9E-0B9F, 0BA3-0BA4, 0BA8-0BAA, 0BAE-0BB5, 0BB7-0BB9, 0BBE-0BC2, 0BC6-0BC8, 0BCA-0BCD
Telugu:	0C01-0C03, 0C05-0C0C, 0C0E-0C10, 0C12-0C28, 0C2A-0C33, 0C35-0C39, 0C3E-0C44, 0C46-0C48, 0C4A-0C4D, 0C60-0C61
Kannada:	0C82-0C83, 0C85-0C8C, 0C8E-0C90, 0C92-0CA8, 0CAA-0CB3, 0CB5-0CB9, 0CBE-0CC4, 0CC6-0CC8, 0CCA-0CCD, 0CDE, 0CE0-0CE1
Malayalam:	0D02-0D03, 0D05-0D0C, 0D0E-0D10, 0D12-0D28, 0D2A-0D39, 0D3E-0D43, 0D46-0D48, 0D4A-0D4D, 0D60-0D61
Thai:	0E01-0E3A, 0E40-0E5B
Lao:	0E81-0E82, 0E84, 0E87-0E88, 0E8A, 0E8D, 0E94-0E97, 0E99-0E9F, 0EA1-0EA3, 0EA5, 0EA7, 0EAA-0EAB, 0EAD-0EAE, 0EB0-0EB9, 0EBB-0EBD, 0EC0-0EC4, 0EC6, 0EC8-0ECD, 0EDC-0EDD
Tibetan:	0F00, 0F18-0F19, 0F35, 0F37, 0F39, 0F3E-0F47, 0F49-0F69, 0F71-0F84, 0F86-0F8B, 0F90-0F95, 0F97, 0F99-0FAD, 0FB1-0FB7, 0FB9
Georgian:	10A0-10C5, 10D0-10F6
Hiragana:	3041-3093, 309B-309C
Katakana:	30A1-30F6, 30FB-30FC
Bopomofo:	3105-312C

CJK Unified Ideographs: 4E00-9FA5

Hangul: AC00-D7A3

Digits: 0660-0669, 06F0-06F9, 0966-096F, 09E6-09EF,
0A66-0A6F, 0AE6-0AEF, 0B66-0B6F, 0BE7-0BEF,
0C66-0C6F, 0CE6-0CEF, 0D66-0D6F, 0E50-0E59,
0ED0-0ED9, 0F20-0F33

Special characters: 00B5, 00B7, 02B0-02B8, 02BB, 02BD-02C1,
02D0-02D1, 02E0-02E4, 037A, 0559, 093D, 0B3D,
1FBE, 203F-2040, 2102, 2107, 210A-2113, 2115,
2118-211D, 2124, 2126, 2128, 212A-2131,
2133-2138, 2160-2182, 3005-3007, 3021-3029

A

abortライブラリ関数	9-45
absライブラリ関数	9-47
acosライブラリ関数	9-20
_align修飾子	2-25
AND ビット演算子 (&)	6-30
ANSI C 規格	
ドキュメント情報	xv
ANSI C 規格制限	1-28
ANSI 規格準拠	B-1
argc	
main関数の実引数	5-16
argv	
main関数の実引数	5-16
ASCII 等価テーブル	C-1
ASCII 文字集合	1-4
asctimeライブラリ関数	9-59
asinライブラリ関数	9-20
assertマクロ	9-3
<assert.h>ヘッダ	9-2
atanライブラリ関数	9-20
atan2ライブラリ関数	9-20
atexitライブラリ関数	9-45
atofライブラリ関数	9-43
atoiライブラリ関数	9-43
atolライブラリ関数	9-43
autoクラス	
定義済みの	2-16
の例	2-16
autoキーワード	
ブロック内の宣言で使用	7-2

B

--_bool_true_false_are_definedマクロ	
□	9-27
_Bool データ型	3-9
ヘッダ・ファイル	9-26
boolマクロ	9-27
break文	
switch文での使用	7-6
定義済みの	7-13
bsearchライブラリ関数	9-46
BUFSIZマクロ	9-29

C

cabsライブラリ関数	9-8
ccoshライブラリ関数	9-5
cacosライブラリ関数	9-4
callocライブラリ関数	9-44
cargライブラリ関数	9-9
caseラベル	7-6
casinhライブラリ関数	9-6
casinライブラリ関数	9-4
catanhライブラリ関数	9-6
catanライブラリ関数	9-4
ccoshライブラリ関数	9-6
ccosライブラリ関数	9-5
ceilライブラリ関数	9-22
cexpライブラリ関数	9-7
cimagライブラリ関数	9-9
clearerrライブラリ関数	9-41
clockライブラリ関数	9-61
clock_t型	9-58
CLOCKS_PER_SECマクロ	9-59

clogライブラリ関数	9-7
Compaq C の ANSI 互換性拡張	
VAX C拡張	E-1
コモン C 拡張	D-1
Compaq C の VAX C 拡張	E-1
Compaq C のコモン C 拡張	D-1
<complex.h>ヘッダ	9-3
_CComplex_Iマクロ	9-3
_CComplex データ型	3-11
complexマクロ	9-3
conjライブラリ関数	9-9
const	
変数宣言での	4-16
ポインタ宣言での	4-16
const型修飾子	
の規則	3-25
の説明	3-24 ~ 3-26
の例	3-24
continue文	7-12
cosライブラリ関数	9-20
coshライブラリ関数	9-21
cpowライブラリ関数	9-8
cprojライブラリ関数	9-9
crealライブラリ関数	9-10
csinhライブラリ関数	9-7
csinライブラリ関数	9-5
csqrtライブラリ関数	9-8
ctanhライブラリ関数	9-7
ctanライブラリ関数	9-5
ctimeライブラリ関数	9-59
<ctype.h>ヘッダ	9-10
C 言語	
演算子の一覧	6-3
C 言語の基本概念	2-1
C 言語の要素	
の文法	1-1
C ライブラリ	
プロトタイプ	5-12

D

__DATE__ あらかじめ定義されたマク	
ロ	8-19
defaultラベル	7-6

#define命令	8-2
defined演算子	8-12
#dictionary命令	E-1
difftimeライブラリ関数	9-61
divライブラリ関数	9-47
div_t型	9-42
do文	7-9

E

EDOMマクロ	9-12
#elif前処理命令	8-9, 8-11
else句	7-4
#else前処理命令	8-9
コモン C 拡張	D-2
#endif前処理命令	8-9
コモン C 拡張	D-2
enumキーワード	4-13
envp	
main 関数の実引数	
VAX C拡張	E-3
コモン C 拡張	D-2
EOFマクロ	9-29
ERANGEマクロ	9-12
errnoマクロ	9-12
<errno.h>ヘッダ	9-12
#error前処理命令	8-19
exitライブラリ関数	9-45
EXIT_FAILUREマクロ	9-42
EXIT_SUCCESSマクロ	9-42
expライブラリ関数	9-21
externクラス	2-18

F

fabsライブラリ関数	9-22
falseマクロ	
stdbool.h	9-27
fcloseライブラリ関数	9-31
feofライブラリ関数	9-41
ferrorライブラリ関数	9-41
fflushライブラリ関数	9-31
fgetcライブラリ関数	9-36
fgetposライブラリ関数	9-39

fgetsライブラリ関数	9-36
FILE型	9-28
__FILE__ あらかじめ定義されたマクロ	
□	8-20
FILENAME_MAXマクロ	9-29
floatキーワード	4-13
floorライブラリ関数	9-22
fmodライブラリ関数	9-22
fopenライブラリ関数	9-32
FOPEN_MAXマクロ	9-29
for文	7-10
forceinline修飾子	2-25
fpos_t型	9-28
fprintfライブラリ関数	9-34
fputcライブラリ関数	9-36
fputsライブラリ関数	9-37
freadライブラリ関数	9-38
freeライブラリ関数	9-44
freopenライブラリ関数	9-33
frexpライブラリ関数	9-21
fscanfライブラリ関数	9-34
fseekライブラリ関数	9-39
fsetposライブラリ関数	9-40
ftellライブラリ関数	9-40
__func__ 宣言済み識別子	8-22
fwriteライブラリ関数	9-38

G

getcライブラリ関数	9-37
getcharライブラリ関数	9-37
getenvライブラリ関数	9-45
getsライブラリ関数	9-37
gmtimeライブラリ関数	9-60
goto文	7-12

H

HUGE_VALマクロ	9-19
-------------	------

I

if文	7-4
if 真の定義	7-5
if 偽の定義	7-5
#if 前処理命令	8-9
defined演算子の使用	8-12
#ifdef 前処理命令	8-9
#ifndef 前処理命令	8-9
_imaginary_Iマクロ	9-3
_imaginary データ型	3-12
imaginaryマクロ	9-3
#include 命令	
モジュール形式	E-2
#include 前処理命令	8-13
#include 行のネスト	8-13
INFINITYマクロ	9-19
inline修飾子	2-20
inline修飾子	2-20
_IOFBFマクロ	9-29
_IOLBFマクロ	9-29
_IONBFマクロ	9-29
isalnumライブラリ関数	9-10
isalphaライブラリ関数	9-10
iscntrlライブラリ関数	9-10
isdigitライブラリ関数	9-11
isgraphライブラリ関数	9-11
islowerライブラリ関数	9-11
ISO C 規格	xv
isprintライブラリ関数	9-11
ispunctライブラリ関数	9-11
isspaceライブラリ関数	9-11
isupperライブラリ関数	9-12
isxdigitライブラリ関数	9-12

J

jmp_buf型	9-23
----------	------

L

L tmpnamマクロ	9-29
labsライブラリ関数	9-47
LC_ALLマクロ	9-14
LC_COLLATEマクロ	9-14
LC_MONETARYマクロ	9-14
LC_NUMERICマクロ	9-14
LC_TIMEマクロ	9-14
LC_TYPEマクロ	9-14
ldexpライブラリ関数	9-21
ldivライブラリ関数	9-47
ldiv_t型	9-42
<limits.h>ヘッダ	9-13
#line前処理命令	8-14
__LINE__あらかじめ定義されたマクロ	8-20
<locale.h>ヘッダ	9-13
localeconvライブラリ関数	9-16
localtimeライブラリ関数	9-60
logライブラリ関数	9-21
log10ライブラリ関数	9-21
long floatキーワード	D-2
longjmpライブラリ関数	9-23
longキーワード	4-12
Lで始まるワイド文字	1-23

M

main 関数	
main_program オプションの使用	E-2
仮引数の引渡し	5-16
の構文	5-16
main_program オプション	E-2
mallocライブラリ関数	9-44
<math.h>ヘッダ	9-19
MB_CUR_MAXマクロ	9-42
mbJenライブラリ関数	9-48
mbstowcsライブラリ関数	9-49
mbtowcライブラリ関数	9-48
memchrライブラリ関数	9-51
memcmpライブラリ関数	9-51
memcpyライブラリ関数	9-50

memmoveライブラリ関数	9-50
memsetライブラリ関数	9-51
mktimeライブラリ関数	9-62
modfライブラリ関数	9-22
#module命令	E-1

N

NANマクロ	9-20
NULLマクロ	
limits.h	9-14
stddef.h	9-28
stdio.h	9-29
stdlib.h	9-42
string.h	9-50
time.h	9-59

O

offsetofマクロ	9-28
-------------	------

P

perrorライブラリ関数	9-41
powライブラリ関数	9-22
#pragma前処理命令	8-15
Pragma演算子	6-23
printfライブラリ関数	9-35
ptrdiff型	9-27
putcライブラリ関数	9-37
putcharライブラリ関数	9-37
putsライブラリ関数	9-38

Q

qsortライブラリ関数	9-46
--------------	------

R

raiseライブラリ関数	9-25
randライブラリ関数	9-44
RAND_MAXマクロ	9-42
realTocライブラリ関数	9-44
registerクラス	2-17
registerキーワード	
ブロック内の宣言で使用	7-2

removeライブラリ関数	9-30
renameライブラリ関数	9-30
__restrict型修飾子	3-29, 4-17
定義	3-33
例	3-35
returnキーワード	
文の構文	7-13
rewindライブラリ関数	9-40

S

scanfライブラリ関数	9-35
SEEK_CURマクロ	9-30
SEEK_ENDマクロ	9-30
SEEK_SETマクロ	9-30
setbufライブラリ関数	9-33
setjmpライブラリ関数	9-23
<setjmp.h>ヘッダ	9-23
setlocaleライブラリ関数	9-14
sig_atomic_t型	9-24
SIG_DFLマクロ	9-24
SIG_ERRマクロ	9-24
SIG_IGNマクロ	9-24
SIGABRTシグナル	9-24
SIGFPEシグナル	9-24
SIGILLシグナル	9-24
signalライブラリ関数	9-24
<signal.h>ヘッダ	9-23
SIGSEGVシグナル	9-24
SIGTERMシグナル	9-24
sinライブラリ関数	9-20
sinhライブラリ関数	9-21
size_t型	
stddef.h	9-27
stdio.h	9-28
stdlib.h	9-42
string.h	9-50
time.h	9-58
sizeof演算子	6-21, B-9
sizeofキーワード	6-21
sprintfライブラリ関数	9-35
sqrtライブラリ関数	9-22
srandライブラリ関数	9-44
sscanfライブラリ関数	9-35

staticクラス	2-18
staticキーワード	
ブロック内の宣言で使用	7-2
<stdarg.h>ヘッダ	9-25
<stdbool.h>ヘッダ	9-26
__STDC__あらかじめ定義されたマクロ	8-20
__STDC_HOSTED__あらかじめ定義されたマクロ	8-21
__STDC_ISO_10646__あらかじめ定義されたマクロ	8-21
__STDC_VERSION__あらかじめ定義されたマクロ	8-21
<stddef.h>ヘッダ	9-27
stderrマクロ	9-30
stdinマクロ	9-30
<stdio.h>ヘッダ	9-28
<stdlib.h>ヘッダ	9-41
stdoutマクロ	9-30
strcatライブラリ関数	9-51
strchrライブラリ関数	9-52
strcmpライブラリ関数	9-52
strcollライブラリ関数	9-52
strcpyライブラリ関数	9-51
strcspnライブラリ関数	9-52
strerrorライブラリ関数	9-54
strftimeライブラリ関数	9-60
<string.h>ヘッダ	9-50
strlenライブラリ関数	9-54
strncatライブラリ関数	9-51
strncmpライブラリ関数	9-52
strncpyライブラリ関数	9-51
strpbrkライブラリ関数	9-53
strrchrライブラリ関数	9-53
strspnライブラリ関数	9-53
strstrライブラリ関数	9-53
strtodライブラリ関数	9-43
strtokライブラリ関数	9-53
strtolライブラリ関数	9-43
strtoulライブラリ関数	9-43
struct lconv型	9-13
struct tm型	9-59
strxfrmライブラリ関数	9-52
switch文	7-5
Compaq C の動作	B-15

switchキーワード
 内部の宣言 7-8
 systemライブラリ関数 9-46

T

tanライブラリ関数 9-20
 tanhライブラリ関数 9-21
 <tgmath.h>ヘッダ 9-54
 timeライブラリ関数 9-62
 <time.h>ヘッダ 9-58
 __TIME__ あらかじめ定義されたマクロ
 □ 8-20
 TMP_MAXマクロ 9-30
 tmpfileライブラリ関数 9-31
 tmpnamライブラリ関数 9-31
 tolowerライブラリ関数 9-12
 toupperマクロ
 関数形式 8-5
 副作用 8-7
 toupperライブラリ関数 9-12
 trueマクロ
 stdbool.h 9-27
 typedef
 構造体タグとの使用 4-44
 の定義 3-43
 typedefキーワード
 宣言で使用 4-44
 __typeof__ 演算子 6-22

U

__unaligned型修飾子 3-28
 __unalignedデータ型修飾子 4-17
 #undef前処理命令 8-2
 ungetcライブラリ関数 9-38

V

va_argマクロ 9-26
 __VA_ARGS__ 識別子 8-5, 8-6
 va_list型 9-25
 va_listマクロ 9-26
 va_startマクロ 9-25

variant struct E-4
 variant union E-4
 VAX Cキーワード E-1
 VAX C組込み関数 E-2
 vfprintfライブラリ関数 9-35
 void型
 定義済みの 3-20
 の使用 3-20
 の例 3-21
 voidポインタ
 定義済みの 4-17
 の使用 3-14
 voidキーワード
 ポインタ 4-17
 voidへのポインタ 3-14
 volatile型修飾子
 の規則 3-27
 の説明 3-27
 vprintfライブラリ関数 9-35
 vsprintfライブラリ関数 9-36

W

wchar_t型
 stddef.h 9-27
 stdlib.h 9-42
 wcstombsライブラリ関数 9-49
 wctombライブラリ関数 9-49
 while文 7-9

X

0x...16 進定数 1-17

ア

アスタリスク演算子 (*) 4-16
 新しい形式の仮引数宣言 5-6
 & (アドレス演算子) 6-18
 アドレス演算子 (&) 6-18
 あらかじめ定義されたマクロ名 8-19

イ

移植性	
char*汎用ポインタ表記	4-17
構造体の境界調整	4-32
ビット・フィールド長	4-34
移植性の問題	
main_program オプション	E-2
ブリプロセッサの処理系	8-1
1 次式	
括弧	6-3
識別子	6-2
定義済みの	6-2
定数	6-2, 6-3
意味エラー	1-3
インクリメント演算子	
++ 後置	6-13
前置	6-16
++ 前置	6-16

ウ

右辺値	2-30
-----	------

エ

エイリアシング	3-30
エスケープ・シーケンス	B-2
エラー	
の種類	1-2
エラー・コード	
ヘッダ・ファイル	9-12
エラー命令	B-19
演算子	
defined	8-12
_Pragma	6-23
sizeof	6-21
__typeof__	6-22
コンマ	6-34
条件	6-31
大括弧	6-8
代入	6-32 ~ 6-33
単項	
アドレス	6-18

演算子

単項 (続き)

インクリメントとデクリメン

ト	6-16
間接参照	6-18
キャスト	6-19
算術否定	6-15
否定	6-16
ビット否定	6-19
定義済みの	1-13
2 項	
加減	6-26
関係	6-28
シフト	6-27
乗除	6-25
等価	6-29
ビット	6-30
論理	6-30
の一覧	6-3
の分類	6-5
の優先順位	6-6
##演算子	B-19
演算子の結合規則	6-7

オ

置換え

「置換」を参照

オペランドの変換	6-41
----------	------

カ

改行文字	1-5
解析	
トップダウン	4-8
解析エラー	1-2
外部オブジェクト定義	B-15
外部結合	2-13
外部宣言	4-8
のスコープ	4-9
外部定義	4-8
加減演算子	6-26
Compaq C の動作	B-10
括弧で囲んだ式	6-3

() (括弧で囲んだ式)	6-3	型変換	6-19
+ (加算演算子)	6-26	型名	2-33
加算に関する逆元	6-15	可変個実引数	
可視性		ヘッダ・ファイル	9-25
定義済みの	2-7	可変個の仮引数並び	5-11
の説明	2-7	可変修飾型	4-3
の例	2-7	可変長配列	4-19, 4-26
型		可変レコード	3-18
clock_t	9-58	仮宣言	
Compaq C	B-6	タグ	4-31
div_t	9-42	仮定義	
FILE	9-28	の説明	2-15
fpos_t	9-28	の定義	4-10
jmp_buf	9-23	の例	2-15
ldiv_t	9-42	仮引数	
ptrdiff_t	9-27	#define前処理マクロでの	8-5
sig_atomic_t	9-24	main関数	5-17
size_t		関数プロトタイプ	5-9
stdint.h	9-27	の規則管理	5-13
stdio.h	9-28	仮引数の引渡し	
stdlib.h	9-42	main関数への	5-16
string.h	9-50	カレント・オブジェクト	4-39
time.h	9-58	関係演算子	6-28
struct lconv	9-13	関数	
struct tm	9-59	C ライブラリ	
va_list	9-25	プロトタイプ	5-12
wchar_t		main	5-16
stdint.h	9-27	アドレス	5-15
stdlib.h	9-42	アドレス渡し	6-11
不完全な	2-10	暗黙の宣言	5-9
ライブラリ	9-1	識別子	6-11
型指定子	B-11	実引数としての	5-15
型修飾子	3-23 ~ 3-42	定義	
Compaq C	B-14	仮引数	5-13
const	3-24	実引数	5-13
__restrict	3-29	実引数の変換	6-10
定義	3-33	定義済みの	5-3
例	3-35	の型	5-2
__unaligned	3-28	の構文	5-10
定義済みの	3-23	の宣言	5-7
の使用	3-23	の定義	
型定義	3-43 ~ 3-44, 4-44	main_program オプション	E-2
型のキャスト	6-19	プロトタイプ	5-9
型汎用数学		変数仮引数並び	5-11
ヘッダ・ファイル	9-54	未宣言の	6-10

関数 (続き)	
呼出し	
の構文	6-10
の定義	5-1
マクロでの	8-7
関数型	
の説明	3-13
の例	3-13
関数スコープ	
定義済みの	2-6
の例	2-6
関数定義	
の構文	5-3
関数のインライン展開	B-20
関数の実引数	
main への	5-16
の変換	6-45
関数プロトタイプ	
定義済みの	5-9
の型変換	5-12
の規則拡張	5-12
のスコープ規則	5-12
関数プロトタイプ・スコープ	
定義済みの	2-6
の例	2-6
* (間接参照演算子)	6-18
間接参照演算子 (*)	6-18
完全型	2-10

キ

記憶域クラス		記憶域存続期間	2-16
auto	2-16	記憶域割り当て	
extern	2-18	順序	4-4
register	2-17	記憶域割付け	
static	2-18	オブジェクトへの	4-4
省略時の設定	2-17	規格準拠処理系	B-1
の型	2-15	規格準拠プログラム	B-1
記憶域クラス指定子	B-11	基本演算子	
記憶域クラス修飾子	2-18	の優先順位	6-6
align	2-25	基本データ型	3-2
__forceinline	2-25	逆元	
inline	2-20	加算に関する	6-15
__inline	2-20	キャスト演算子	6-19
		Compaq C の動作	B-9
		() (キャスト演算子)	6-19
		旧形式の仮引数宣言	
		の例	5-6
		プロトタイプ形式との比較	5-9
		旧形式の関数宣言	
		プロトタイプ形式との併用	2-11
		境界調整	
		定義済み定数	2-26
		境界調整されていないデータ	3-28
		境界調整単位	3-17
		共通定義	
		ヘッダ・ファイル	9-27
		共用体	
		の初期化	4-38
		の宣言	4-29
		変形集合体	E-4
		メンバ	
		の参照	6-12
		共用体型	
		の説明	3-18
		共用体指定子	B-11
		共用体の宣言	
		の構文	4-28, 4-29
		キーワード	
		align	2-25
		break文	7-13
		caseラベル	7-6
		continue文	7-12
		defaultラベル	7-6
		do文	7-9
		else句	7-4

キーワード (続き)

enum	4-13
for文	7-10
__forceinline	2-25
_goto文	7-12
if文	7-4
inline	2-20
__inline	2-20
_return文	7-13
sizeof	6-21
switch文	7-5
VAX C	B-4
void	4-17
while文	7-9
定義済みの	1-11
の一覧	1-11
の使用	1-11

ク

空 (#) 前処理命令	8-19
空白	
の使用	1-2
文字	1-4
空文	7-3
空ポインタ	
自動初期化	4-17
定義済みの	3-14
等価演算子とともに使用	6-33
区切り記号	1-14 ~ 1-15
区切り記号 (...)	5-11
繰返し文	7-8 ~ 7-11

ケ

継続

文字列	1-5
文字列の終わり	1-16
論理行	1-5

継続文字

#defineでの	8-4
-----------	-----

結合

外部	2-13
内部	2-13
なし	2-13

結合 (続き)

の型	2-13
の決定	2-13
の例	2-14
結合プラグマ	B-21
言語の要素	1-1
現在印字位置	1-30
限定ポインタ	3-29
定義	3-33
例	3-35

コ

合成型

定義済みの	2-12
の条件	2-12

構成子式

構造体

順方向参照	4-31
初期化	4-39
の初期化	4-36
の宣言	4-29 ~ 4-32
ビット・フィールド	4-34
変形集合体	E-4
メンバ	

の参照	6-12
-----	------

構造体型

構造体指定子

. (構造体と共用体演算子)

構造体と共用体の相違点

構造体と共用体の類似点

構造体の境界調整

OpenVMS Alpha 上	B-12
OpenVMS VAX 上の	B-13 ~ B-14
Tru64 UNIX 上	B-12
の説明	3-17, B-12

構造体の宣言

の構文	4-28, 4-29
-----	------------

-> (構造体または共用体ポインタ演算

子)	6-12
----	------

構造体メンバ

の宣言	4-29
-----	------

後置インクリメント演算子

++ (後置インクリメント演算子)

後置式	
インクリメント演算子	6-13
関数呼出し	6-10
共用体の参照	6-12
構造体の参照	6-12
デクリメント演算子	6-13
配列の参照	6-8
後置デクリメント演算子	6-13
-- (後置デクリメント演算子)	6-13
構文	
main関数	5-16
構文エラー	1-2
互換型	
カテゴリ	2-10
定義済みの	2-10
互換性のない型	2-12
コマンド行実引数	5-16
コメント	1-9
コンパイル単位	
定義済みの	1-3
データ共有	2-3
の説明	2-3
, (コンマ演算子)	6-34
コンマ演算子 (,)	6-34
の優先順位	6-6
 サ	
最初の宣言	4-10
左辺値	2-30
3 項演算子	1-13
算術	
否定演算子	6-15
複素数	9-3
ヘッダ・ファイル	9-19
算術型	3-3
算術変換	
通常の	6-41
3 文字表記	1-6

シ

式	
1 次	
括弧	6-3
識別子	6-2
定義済みの	6-2
定数	6-2, 6-3
の構文	6-2
関係	6-28
構成子	6-36
後置	
インクリメント演算子	6-13
関数呼出し	6-10
共用体の参照	6-12
構造体の参照	6-12
デクリメント演算子	6-13
の構文	6-8
配列の参照	6-8
コンマ	6-34
条件	6-31
代入	6-32
単項	
Pragma	6-23
sizeof	6-21
typeof	6-22
アドレス指定の	6-18
インクリメントおよびデクリメン	
ト	6-16
キャスト	6-19
算術否定	6-15
否定	6-16
ビット否定	6-19
定数	6-34
2 項	
加減	6-26
関係	6-28
シフト	6-27
乗除	6-25
等価	6-29
ビット	6-30
論理	6-30
の評価順序	2-8
複合リテラル	6-36

式 (続き)	
文としての	7-3
識別子	
Compaq C	B-5
結合	B-5
有効文字	1-9
宣言済み	
__func__	8-22
定義済みの	1-7
の規則	1-8
字句解析エラー	1-2
シグナル処理	
ヘッダ・ファイル	9-23
実行文字集合	
定義済みの	1-3
の一覧	1-5
システム識別のあらかじめ定義されたマク	
ロ	8-21
実引数	
#define前処理マクロでの	8-5
値渡し	5-13
可変個	
ヘッダ・ファイル	9-25
関数に対する	
の変換	5-14
関数の変換	6-45
関数プロトタイプ	5-9
コマンド行	5-16
実引数としての関数	5-15
実引数としての配列	5-15
の規則管理	5-13
の変換	5-14
実引数拡張	
省略時の	6-12
自動記憶域存続期間	2-16
シフト演算子 (<<と>>)	6-27
指名子	4-39, 4-40
集合体配列	4-19
「大括弧演算子 ([])」を参照	
集合体変形	E-4
修飾子	
記憶域クラス	2-18
集成型	3-13
16 進定数	1-17
順方向参照	
定義済みの	2-27
の例	2-27
?:(条件演算子)	6-31
条件演算子 (?:)	
定義済みの	6-31
の優先順位	6-6
条件付きコンパイル	8-9 ~ 8-12
条件付き取込み	B-15
条件文	7-4 ~ 7-8
* (乗算演算子)	6-25
乗算演算子 (*)	6-25
乗除演算子	B-10
% (剰余演算子)	6-25
剰余演算子 (%)	6-25
省略記号	
プロトタイプでの	5-11
省略時の拡張変換	5-14
省略時の実引数拡張	6-12
初期化	4-5
C99 規格	4-39
Compaq C の動作	B-15
暗黙の	4-5
一般の	4-5
共用体	4-38
構造体	4-36, 4-39
の規則	4-5
配列	4-20, 4-22, 4-39
初期値	
の構文	4-5
/ (除算演算子)	6-25
除算演算子 (/)	6-25
診断	B-2
診断メッセージ	
ヘッダ・ファイル	9-2
0...8 進定数	1-17
16 進浮動小数点定数	1-21
ス	
数学	
複素数	9-3
数学, 型汎用	
ヘッダ・ファイル	9-54

数値エスケープ・シーケンス . . .	1-26 ~ 1-27
数値制限	
定義済みの	1-29
数量的限界	B-3
スコープ	2-4

セ

制御文字	1-5
制限	
ANSI	1-28
数値の	1-29
定義済みの	1-28
ヘッダ・ファイル	9-13
翻訳	1-28
整数	
浮動小数点への変換	B-8
整数オブジェクトの宣言	4-12
整数定数	
定義済みの	1-17
の規則	1-17
8進数の	B-6
整数データ型	
の宣言	4-12
静的記憶域存続期間	2-16
宣言	
C ライブラリ・プロトタイプ	5-12
一般的な構文	4-2
型定義	4-44
関数	5-7
関数プロトタイプ	5-9
共用体	4-29
構造体	4-29
集合体の配列	4-19
タグの仮宣言	4-31
の構文規則	4-3
の書式	4-11
の例	4-3
ブロック内	7-2
列挙型	4-13
宣言子	B-14
選択文	7-4 ~ 7-8
前置インクリメント演算子	6-16
++ (前置インクリメント演算子)	6-16

前置デクリメント演算子	6-16
-- (前置デクリメント演算子)	6-16

ソ

添字付け	6-8
ソース・ファイルの取込み	B-16
ソース文字集合	
記憶域	3-9
定義済みの	1-3

タ

[] (大括弧演算子)	6-8
大括弧演算子 ([])	6-8
代入演算子	6-32 ~ 6-33
の優先順位	6-6
= (代入演算子)	6-32
+= (代入演算子)	6-32
-= (代入演算子)	6-32
*= (代入演算子)	6-32
タグ	
の仮宣言	4-31
の説明	2-28
の宣言構文	4-43
の例	2-28
多次元配列	4-22
の添字	3-15
多バイト文字	
Compaq C	B-2
定義済みの	1-24
単項演算子	
定義済みの	1-13
の優先順位	6-6
単項式	
Pragma	6-23
sizeof	6-21
typeof	6-22
アドレス --	6-18
インクリメントとデクリメント	6-16
間接参照	6-18
キャスト	6-19
算術否定	6-15
否定	6-16
ビット否定	6-19

単項マイナス演算子	6-15
単純オブジェクト	
省略時の初期化	4-12
の初期化	4-11
の宣言書式	4-11

チ

置換	
#include命令中の	8-14
マクロ	8-2
規則	8-6
抽象宣言子	
キャスト	6-20
定義済みの	2-33
の例	2-33

ツ

通常の算術変換	6-41
---------	------

テ

定義	
オブジェクト	4-4
関数	5-3
定義済みの境界調整定数	2-26
ディジグネーション	4-40
初期化子並びでの	4-39
定数	
#defineマクロでの識別子	8-4
整数	1-17
定義済みの	1-17
浮動小数点	1-20
16進	1-21
文字	1-22
列挙	1-27
定数式	
アドレス定数	6-36
算術	6-35
定義済みの	6-34
汎整数	6-35
デクリメント演算子	
-- 後置	6-13
前置	6-16

デクリメント演算子 (続き)

-- 前置	6-16
データ型	
_Bool	3-9
関数	5-2
関数プロトタイプ	5-9
基本	3-2
の一覧	3-2
の概要	3-1
のサイズ	3-5
の範囲	3-5
の文字	3-9
汎整数	3-6
複素数	3-11
浮動小数点	3-11
変換	6-40
データ型の拡張	6-40

ト

== (等価演算子)	6-29
等価演算子 (=)	6-29
<= (等価左不等号演算子)	6-28
等価左不等号演算子 (<=)	6-28
>= (等価右不等号演算子)	6-28
等価右不等号演算子 (>=)	6-28
トークン	1-1
トークンの置換	8-2
ドット演算子 (.)	6-12
飛越し, 非ローカル	
ヘッダ・ファイル	9-23
飛越し文	7-11 ~ 7-13

ナ

内部結合	2-13
内部宣言	4-8
名前空間	
定義済みの	2-31
の型	2-31

二

2 項演算子	
加減	6-26
関係	6-28
シフト	6-27
乗除	6-25
定義済みの	1-13
等価	6-29
の優先順位	6-6
ビット	6-30
論理	6-30
2 文字表記	1-7
入出力	
ヘッダ・ファイル	9-28

ヌ

ヌル文字	1-5
------	-----

ネ

ネストされた#include行の限界	8-13
--------------------	------

ハ

配列	
可変長	4-26
サイズの決定	3-15
式としての	6-8
実引数としての	5-15
初期化	4-39
の参照	6-8
の初期化	4-22
の宣言	4-19
配列型	
の説明	3-15
配列記憶域	
行優先順序	4-22
配列宣言	
の構文	4-19
配列添字	3-16
配列ポインタ	4-25

バックスラッシュ継続文字

#defineでの	8-4
バックスラッシュと改行の継続文字	1-5
8 進数字の 8 と 9	B-6, D-1
8 進定数	1-17
派生型の一覧	3-13
汎整数拡張	6-41
値保存	B-7
符号なし保存	B-7
汎整数型	3-6
の説明	3-6 ~ 3-7

ヒ

<< (左シフト演算子)	6-27
< (左不等号演算子)	6-28
左不等号演算子 (<)	6-28
日付と時刻	
ヘッダ・ファイル	9-58
否定	
算術	6-15
論理	6-16
& (ビット AND 演算子)	6-30
(ビット OR 演算子)	6-30
ビット OR 演算子 ()	6-30
^ (ビット XOR 演算子)	6-30
ビット XOR 演算子 (^)	6-30
!= (非等価演算子)	6-29
非等価演算子 (!=)	6-29
ビット演算子	6-30
ビット単位のシフト演算子	B-11
ビット否定演算子 (~)	6-19
ビット・フィールド	
OpenVMS VAX 上の境界調整	B-14
ULTRIX RISC 上の境界調整	B-12
コモン C のデータ型	D-1
長さ 0 の	4-35
名前のない	4-35
の規則	4-35
の作成	4-35
の宣言	4-34
の宣言構文	4-34
評価順序	
実引数並びの	5-13

評価順序点	2-8
標準ヘッダ・ファイル	9-1
ピリオド演算子 (.)	6-12
非ローカル飛越し	
ヘッダ・ファイル	9-23

フ

ファイル		浮動小数点型	3-11
ヘッダ	9-1	の一覧	3-11
ファイル・スコープ		浮動小数点定数	
定義済みの	2-4	型	1-21
の例	2-5	16進	1-21
ファイルの取込み	8-13	定義済みの	1-20
Cライブラリ・プロトタイプ	5-12	表記	1-22
不完全型		ブAGMA	
タグによる作成	2-29	VAX C	B-21
定義済みの	2-9	命令	B-20
の例	2-10	ボール型とボール値	
不完全配列宣言		ヘッダ・ファイル	9-26
の使用	4-20	ブロック	
の例	3-15	定義済みの	2-2
複文	7-2	の例	2-2
複合代入演算子	B-7	ブロック文	7-2
複合リテラル	6-36	ブロック・スコープ	
副作用		定義済みの	2-5
定義済みの	2-8	の例	2-5
の例	2-9	プロトタイプ	
マクロ内の	8-7	Cライブラリ関数の	5-12
複素数算術		定義済みの	5-9
ヘッダ・ファイル	9-3	プロトタイプ形式の仮引数宣言	5-6
複素数データ型	3-11	プロトタイプ形式の関数宣言	
符号付き整数	B-8	旧形式との併用	2-11
符号なし整数	B-8	定義済みの	5-9
符号なし汎整数型	3-8	文	
符号なし保存規則		break	
VAX Cモード	E-4	case文で使用	7-6
コモン C モード	D-3	定義済みの	7-13
浮動小数点		continue	7-12
整数への変換	B-8	default	7-6
データ型		do	
の精度	4-13	繰返し文	7-8
の宣言	4-13	定義済みの	7-9
浮動小数点オブジェクトの宣言	4-13	for	
		繰返し文	7-8
		定義済みの	7-10
		goto	7-12
		if	7-4
		like	7-8
		return	7-13
		switch	7-5
		while	7-9
		空	7-3

文 (続き)

繰返し	7-8 ~ 7-11
式	7-3
条件	7-4 ~ 7-8
選択	7-4 ~ 7-8
飛越し	7-11 ~ 7-13
複合またはブロック	7-2
ラベル	7-1

へ

ヘッダ・ファイル

<assert.h>	9-2
<complex.h>	9-3
<ctype.h>	9-10
<errno.h>	9-12
<limits.h>	9-13
<locale.h>	9-13
<math.h>	9-19
<setjmp.h>	9-23
<signal.h>	9-23
<stdarg.h>	9-25
<stdbool.h>	9-26
<stddef.h>	9-27
<stdio.h>	9-28
<stdlib.h>	9-41
<string.h>	9-50
<time.h>	9-58
定義済みの	1-27, 9-1
変換	6-40
関数の実引数	6-10
関数プロトタイプ	6-45
キャスト演算子付き	6-19
算術データ型	6-40
通常の算術	6-41
データ型の	6-40
変形共用体	E-4
Compaq C の動作	B-12
変形構造体	
Compaq C の動作	B-12
定義済みの	E-4
変更可能な左辺値	2-31
変数	
の初期化	4-11

ホ

ポインタ

void	4-17
インクリメント演算子 (++) を使 用	6-17
空	4-17
単項演算子	6-18
の宣言	4-16
配列への	4-25
ポインタ型	
の説明	3-14
被参照型	3-14
ポインタ宣言	
の構文	4-16
ポインタの間接参照	6-18
ポインタの初期化	4-18
ポインタのスケーリング	4-25
ポインタ変換	B-9
補数演算子 (~)	6-19
ホスト環境	B-2
翻訳	
C コードの	2-32
段階	2-32
翻訳限界	B-3
翻訳上の制限	1-28

マ

マイナス演算子

単項	6-15
前処理	
の説明	2-32
前処理演算子	
#	8-7
##	8-9
前処理命令	
#define	8-2
#elif	8-9
#else	8-9
#endif	8-9
#error	8-19
#if	8-9
#ifdef	8-9

前処理命令 (続き)

#ifndef	8-9
#include	
定義済みの	8-13
マクロ置換	8-14
#line	8-14
#pragma	8-15
#undef	8-2
空 (#)	8-19
マクロ	
assert	9-3
bool	9-27
__bool_true_false_are_defined	
	9-27
BUFSIZ	9-29
CLOCKS_PER_SEC	9-59
complex	9-3
Complex_I	9-3
EDOM	9-12
EOF	9-29
ERANGE	9-12
errno	9-12
EXIT_FAILURE	9-42
EXIT_SUCCESS	9-42
false	
stdbool.h	9-27
FILENAME_MAX	9-29
FOPEN_MAX	9-29
HUGE_VAL	9-19
imaginary	9-3
Imaginary_I	9-3
#include命令中での置換	8-14
INFINITY	9-19
_IOFBF	9-29
_IOLBF	9-29
_IONBF	9-29
L_tmpnam	9-29
LC_ALL	9-14
LC_COLLATE	9-14
LC_MONETARY	9-14
LC_NUMERIC	9-14
LC_TIME	9-14
LC_TYPE	9-14
MB_CUR_MAX	9-42
NaN	9-20

マクロ (続き)

NULL	
limits.h	9-14
stddef.h	9-28
stdio.h	9-29
stdlib.h	9-42
string.h	9-50
time.h	9-59
offsetof	9-28
RAND_MAX	9-42
SEEK_CUR	9-30
SEEK_END	9-30
SEEK_SET	9-30
SIG_DFL	9-24
SIG_ERR	9-24
SIG_IGN	9-24
stderr	9-30
stdin	9-30
stdout	9-30
TMP_MAX	9-30
true	
stdbool.h	9-27
va_arg	9-26
va_list	9-26
va_start	9-25
あらかじめ定義された	
__DATE__	8-19
__FILE__	8-20
__LINE__	8-20
__STDC__	8-20
__STDC_HOSTED__	8-21
__STDC_ISO_10646__	8-21
__STDC_VERSION__	8-21
__TIME__	8-20
システム識別	8-21
定義済みの	8-19
参照	8-6
置換	8-2
定義	8-2
#演算子	8-7
##演算子	8-9
起こりうる副作用	8-7
オブジェクト形式	8-4
仮引数の命名	8-5
関数形式	8-5

マクロ	
定義 (続き)	
の取消し	8-2
ライブラリ	9-1
マクロ展開, プラグマ内	8-16
マクロ名	
一覧	B-16

ミ

>> (右シフト演算子)	6-27
> (右不等号演算子)	6-28
右不等号演算子 (>)	6-28

ム

無意味なタグ宣言	
の例	2-29

メ

命令	
「前処理命令」を参照	
メンバ	
共用体	B-9
構造体	B-9
変形集合体	E-4

モ

文字	
データ型	
オブジェクト	4-12
文字列	4-19
「配列」を参照	
文字エスケープ・シーケンス	
の一覧	1-25
文字オブジェクトの宣言	4-13
文字型	3-9
文字集合	
定義済みの	1-3
文字処理	
ヘッダ・ファイル	9-10
文字定数	
Compaq C に固有の	B-6

文字定数 (続き)	
定義済みの	1-22
の値	1-23
文字表示	
定義済みの	1-30
文字列	3-10
定義済みの	1-15
文字列処理	
ヘッダ・ファイル	9-50
文字列リテラル	
Compaq C	B-6
定義済みの	1-15
の修正	4-19
の例	1-16

ヤ

矢印演算子 (->)	6-12
------------	------

ユ

有効文字	
識別子	1-9
優先順位	
演算子	6-6
定義済みの	1-13
の説明	6-5
ユーザ定義関数	
「関数」を参照	
ユーティリティ	
ヘッダ・ファイル	9-41
ユニバーサル・キャラクタ名	1-9, F-1
緩やかなポインタおよび整数の互換性	
VAX C	E-1
コモン C	D-1

ラ

ライブラリ関数	9-1 ~ 9-62
abort	9-45
abs	9-47
acos	9-20
asctime	9-59
asin	9-20
atan	9-20

ライブラリ関数 (続き)

atan2	9-20
atexit	9-45
atof	9-43
atoi	9-43
atol	9-43
bsearch	9-46
cabs	9-8
cacos	9-4
cacosh	9-5
calloc	9-44
carg	9-9
casin	9-4
casinh	9-6
catan	9-4
catanh	9-6
ccos	9-5
ccosh	9-6
ceil	9-22
cexp	9-7
cimag	9-9
clearerr	9-41
clock	9-61
clog	9-7
conj	9-9
cos	9-20
cosh	9-21
cpow	9-8
cproj	9-9
creal	9-10
csin	9-5
csinh	9-7
csqrt	9-8
ctan	9-5
ctanh	9-7
ctime	9-59
difftime	9-61
div	9-47
exit	9-45
exp	9-21
fabs	9-22
fclose	9-31
feof	9-41
ferror	9-41
fflush	9-31

ライブラリ関数 (続き)

fgetc	9-36
fgetpos	9-39
gets	9-36
floor	9-22
fmod	9-22
fopen	9-32
fprintf	9-34
fputc	9-36
fputs	9-37
fread	9-38
free	9-44
freopen	9-33
frexp	9-21
fscanf	9-34
fseek	9-39
fsetpos	9-40
ftell	9-40
fwrite	9-38
getc	9-37
getchar	9-37
getenv	9-45
gets	9-37
gmtime	9-60
isalnum	9-10
isalpha	9-10
iscntrl	9-10
isdigit	9-11
isgraph	9-11
islower	9-11
isprint	9-11
ispunct	9-11
isspace	9-11
isupper	9-12
isxdigit	9-12
labs	9-47
ldexp	9-21
ldiv	9-47
localeconv	9-16
localtime	9-60
log	9-21
log10	9-21
longjmp	9-23
malloc	9-44
mblen	9-48

ライブラリ関数 (続き)

mbstowcs	9-49
mbtowc	9-48
memchr	9-51
memcmp	9-51
memcpy	9-50
memmove	9-50
memset	9-51
mktime	9-62
modf	9-22
perror	9-41
pow	9-22
printf	9-35
putc	9-37
putchar	9-37
puts	9-38
qsort	9-46
raise	9-25
rand	9-44
realloc	9-44
remove	9-30
rename	9-30
rewind	9-40
scanf	9-35
setbuf	9-33
setjmp	9-23
setlocale	9-14
signal	9-24
sin	9-20
sinh	9-21
sprintf	9-35
sqrt	9-22
srand	9-44
sscanf	9-35
strcat	9-51
strchr	9-52
strcmp	9-52
strcoll	9-52
strcpy	9-51
strcspn	9-52
strerror	9-54
strftime	9-60
strlen	9-54
strncat	9-51
strncmp	9-52

ライブラリ関数 (続き)

strncpy	9-51
strpbrk	9-53
strrchr	9-53
strsfrm	9-52
strspn	9-53
strstr	9-53
strtod	9-43
strtok	9-53
strtol	9-43
strtoul	9-43
system	9-46
tan	9-20
tanh	9-21
time	9-62
tmpfile	9-31
tmpnam	9-31
tolower	9-12
toupper	9-12
ungetc	9-38
vfprintf	9-35
vprintf	9-35
vsprintf	9-36
wcstombs	9-49
wctomb	9-49

ラベル付き文

case	7-6
定義済みの	7-1

リ

リテラル

複合	6-36
----------	------

ル

ルーチン

ライブラリ	9-1
-------------	-----

ループ文

「繰返し文」を参照

レ

列挙型	
の説明	3-22
の例	3-22
列挙型指定子	B-14
列挙定数	
定義済みの	1-27
の型	3-22
の構文	4-13
列挙データ型	
の宣言	4-13

ロ

ローカル化	
ヘッダ・ファイル	9-13
論理	
算術演算子	6-30

否定演算子	6-16
論理偽の定義	7-5
論理行	1-5
! (論理式)	6-16
論理真の定義	7-5
&& (論理積演算子)	6-30
(論理和演算子)	6-30

ワ

ワイド文字	
定義済みの	1-23
の使用	1-4
ワイド文字型	3-10
ワイド文字定数	
定義済みの	1-23
の例	1-23
ワイド文字の文字列	1-23
割付け	
記憶域	4-4

Tru64 UNIX オペレーティング・システム
Compaq C 言語リファレンス・マニュアル

2002 年 11 月 発行

日本ヒューレット・パッカード株式会社

〒140-8641 東京都品川区東品川 2-2-24 天王洲セントラルタワー

電話 (03)5463-6600 (大代表)

AA-RK3VE-TE

