LAPIS SEMICONDUCTOR

SQ003023E092

CCU8

ランゲージ リファレンス

プログラム開発支援ソフトウェア

2011 年 10 月発行

ご注意

本資料の一部または全部をラピスセミコンダクタの許可なく、転載・複写することを堅くお断りします。

本資料の記載内容は改良などのため予告なく変更することがあります。

本資料に記載されている内容は製品のご紹介資料です。ご使用にあたりましては、別途仕様書を必ずご請求のうえ、ご確認ください。

本資料に記載されております応用回路例やその定数などの情報につきましては、本製品の標準的な動作や使い方を説明するものです。したがいまして、量産設計をされる場合には、外部諸条件を考慮していただきますようお願いいたします。

本資料に記載されております情報は、正確を期すため慎重に作成したものですが、万が一、当該情報の誤り・誤植に起因する損害がお客様に生じた場合においても、ラピスセミコンダクタはその責任を負うものではありません。

本資料に記載されております技術情報は、製品の代表的動作および応用回路例などを示したものであり、ラピスセミコンダクタまたは他社の知的財産権その他のあらゆる権利について明示的にも黙示的にも、その実施または利用を許諾するものではありません。上記技術情報の使用に起因して紛争が発生した場合、ラピスセミコンダクタはその責任を負うものではありません。

本資料に掲載されております製品は、一般的な電子機器（AV 機器、OA 機器、通信機器、家電製品、アミューズメント機器など）への使用を意図しています。

本資料に掲載されております製品は、「耐放射線設計」はなされておりません。

ラピスセミコンダクタは常に品質・信頼性の向上に取り組んでおりますが、種々の要因で故障することもあり得ます。

ラピスセミコンダクタ製品が故障した際、その影響により人身事故、火災損害等が起こらないようご使用機器でのディレーティング、冗長設計、延焼防止、フェイルセーフ等の安全確保をお願いします。定格を超えたご使用や使用上の注意書が守られていない場合、いかなる責任もラピスセミコンダクタは負うものではありません。

極めて高度な信頼性が要求され、その製品の故障や誤動作が直接人命を脅かしあるいは人体に危害を及ぼすおそれのある機器・装置・システム（医療機器、輸送機器、航空宇宙機、原子力制御、燃料制御、各種安全装置など）へのご使用を意図して設計・製造されたものではありません。上記特定用途に使用された場合、いかなる責任もラピスセミコンダクタは負うものではありません。上記特定用途への使用を検討される際は、事前にローム営業窓口までご相談願います。

本資料に記載されております製品および技術のうち「外国為替及び外国貿易法」に該当する製品または技術を輸出する場合、または国外に提供する場合には、同法に基づく許可が必要です。

Windows は、米国 Microsoft Corporation の米国およびその他の国における登録商標です。また、その他の製品名や社名などは、一般に商標または登録商標です。

ラピスセミコンダクタ株式会社

〒193-8550 東京都八王子市東浅川町 550 番地 1

http://www.lapis-semi.com/jp/

目　次

1. プリプロセッサ

1.1 概要

CCU8 では、/LP または/PC オプションを付けて呼び出すと、テキストをコンパイルしないで処理することができます。/LP または/PC オプションの呼び出しを行うと、CCU8 は、ソースファイルであるテキストを操作するためのテキストプロセッサのように動作します。

プリプロセッサには、次のような機能があります。

1.マクロの置換

2.条件付きコンパイル

3.ファイルのインクルード

4.行制御

5.エラーの生成

6.ミックスドランゲージプログラミング

7.その他の実装に関する動作(プログラムを使用)

8.トリグラフ文字列(連続する 3 文字の文字列)の置換

#で始まる行は、先行文字として空白文字を持つ場合がありますが、プリプロセッサとデータの送受信を行います。このような行の構文は、その言語で記述された他の部分とは無関係です。行の境界は重要です。ファイルの終了は、前処理指令行で発生させてはいけません。プリプロセッサへの指令は、ソースファイル中の任意の場所に記述してかまいません。ただし、効果はその指令を記述した後の部分に対してのみ適用されます。

1.2 翻訳フェーズ

前処理は、次の 4 段階の翻訳フェーズにおいて、所定の方法で行われます。

1. トリグラフ文字列を適切な 1 文字の内部表現で置換する。
2. 改行文字と直前のバックスラッシュを削除して物理的ソース行を論理的ソース行に連結する。
3. ソースファイルを前処理トークンと空白文字列(コメントを含む)とに分割する。
4. 前処理指令を実行し、マクロを展開する。#include 前処理指令では、指定されたヘッダーファイルまたはソースファイルに対してフェーズ 1～4 を循環的に実行する。

1.2.1 トリグラフ文字列

次に列挙する連続する 3 文字の文字列(トリグラフシーケンス)は、ソースファイル中で出現するたびに、それぞれ対応する 1 文字に置換されます。

トリグラフ文字列	置換されるテキスト
??=	#
??(	[
??/	\
??)	]
??’	^
??<	{
??!	|
??>	}
??-	~

上記トリグラフのいずれにも該当しない文字列の先頭にある「?」は、変更されません。

例 1.1

INPUT:

```
main ()
??<
??>
```

OUTPUT:

```
main ()
{
}
```

1.2.2 行連結

改行文字と直前のバックスラッシュが削除され、改行文字以降の行は前の行の続きとして解釈されます。

1.3 マクロ

1.3.1 概要

マクロは、ユーザによるトークン文字列を識別名で簡略化するものです。指定された識別名は、そのソースファイル中ではトークン文字列を表現するものとして使用可能になります。

次の 2 つの前処理指令を使用すると、簡単にマクロを定義/未定義することができます。

a) #define

b) #undef

マクロの展開は、マクロ名を対応するトークン文字列で置換するテキスト処理の 1 つです。

パラメータは、マクロに転送する引数を表現するものとしても定義されます。引数を持つマクロの置換用テキストは、別の呼び出しに変えることができます。次の a)と b)は、置換処理に効力を持つ専用演算子です。

a) stringizing (#)

b) token pasting (##)

1.3.2 マクロ定義

1.3.2.1 マクロ定義(パラメータなし)

構文:

***# define** identifier token_sequence*

#define 指令では、プリプロセッサが、以降に記述された識別名(identifier)を指定されたトークンの順に置換します。

トークン文字列の前後にある空白文字は切り捨てられます。

マクロ名は C の識別名として有効でなければなりません。トークン文字列(token_sequence)は、置換されるテキストを表現します。

例 1.2

```
# define ABC        1 + 2
マクロ呼び出し       置換されるテキスト
ABC + 3             1 + 2 + 3
fn(ABC)             fn(1 + 2)
```

1.3.2.2 マクロ定義(パラメータ付き)

構文:

***# define** identifier([parameter_list]) token_sequence*

マクロ定義では、識別名(identifier)と開始括弧「(」の間に空白がない場合はパラメータの定義があるものと解釈されます。

パラメータリスト(parameter_list)は任意指定です。存在する場合は、1 つ以上のパラメータで構成されます。パラメータリストは、括弧で囲まれていなければなりません。個々のパラメータは、パラメータリスト内では、一意な識別子でなければなりません。隣のパラメータとの間は、コンマで区切ります。

パラメータは、トークン文字列(token_sequence)中の引数を置換しなければならない場所に現れます。ただし、同じパラメータを 1 つのトークン文字列中に複数回記述することができます。

トークン文字列の前後の空白文字は切り捨てられます。

例 1.3

```
# define ABC(x,y)    x + y
```

マクロ呼び出し	置換されるテキスト
ABC (1,2)	1 + 2
ABC (2,x)	2 + x

1.3.2.3 マクロ処理演算子

1.3.2.3.1 Stringizer

演算子シンボル: #

構文:

 # parameter

stringizer は、パラメータ付きマクロの定義内でのみ使用します。トークン文字列内で記述します。パラメータ(parameter)は stringizer の後ろに記述します。

展開中、引数は引用符で囲まれ、文字列リテラルとして扱われます。

バックスラッシュ「\」は、各「"」および文字定数または文字列リテラルとしての「\」の前に挿入されます (区切り文字としての「"」を含みます)。

例 1.4

```
# define A(b)        #b
# define X(y)        (#y "\n")
```

マクロ呼び出し	置換されるテキスト
A(1)	"1"
X(abc)	("abc" "\n")
A("a\c")	"\"a\\c\""
A("abcde\n")	"\"abcde\\n\""

1.3.2.3.2 Token Paster

演算子シンボル: ##

構文:

token ## token

Token Paster は、マクロの定義内で使用されます。パラメータの有無は無関係です。

Token Paster 演算子は、隣り合うトークン(token)を間にある空白文字を削除しながら連結して新しいトークンを構成します。

Token Paster は、トークン文字列の先頭および末尾では使えません。

例 1.5

```
# define A(b,c)     b ## c
# define X(a)       a ## 1
# define Y(a)       1 ## a
# define ONE        12 ##4
```

マクロ呼び出し	置換されるテキスト
A(1,2)	12
X(34)	341
Y(43)	143
ONE	124

1.4 マクロの展開

1.4.1 マクロの展開(パラメータなし)

パラメータのないマクロとして定義された識別名に従属するインスタンスは、プリプロセッサによって指定されたトークンの順に置換されます。

例 1.6

```
# define ONE              1
# define TWO              2
```

マクロ呼び出し	置換されるテキスト
x = ONE + TWO + ONE	x = 1 + 2 + 1

置換されたトークン文字列は、定義されている識別名が存在するかどうかを確認するために繰り返し走査されます。

例 1.7

```
# define ONE                THREE
# define TWO                2
# define THREE              3
マクロ呼び出し              置換されるテキスト
x = ONE + TWO + THREE       x = 3 + 2 + 3
```

置換された識別名が再走査中に再び現れた場合、その識別名は置換されません。再帰を防ぐために、変更されずにそのまま残ります。

例 1.8

```
# define ONE                TWO
# define TWO                THREE
# define THREE              TWO
マクロ呼び出し              置換されるテキスト
x = ONE + TWO + THREE       x = TWO + TWO + THREE
```

展開中、連続する空白文字は、1 文字の空白文字に置換されます。

例 1.9

```
# define ABCD       a + b + c+d
# define XYZ        a/* abcde */+ 2
マクロ呼び出し      置換されるテキスト
ABCD                a + b + c+d
XYZ                 a + 2
```

マクロ識別名が引用符で囲まれている場合は、マクロ呼び出しとはみなされません。

例 1.10

```
# define ONE        1
マクロ呼び出し      置換されるテキスト
“ONE”               “ONE”
```

1.4.2 マクロの展開(パラメータ付き)

パラメータを持つマクロとして定義された識別名(identifier)は、次のように記述して呼び出します。

identifier[white space]([actual_argument_list])

例 1.11

```
# define ADD(a,b)                 a + b
# define MUL(a,b)                 (a * b)
マクロ呼び出し                    置換されるテキスト
x=MUL(23,43) - ADD(12,400)        x=(23 * 43) - 12 + 400
```

マクロ呼び出しの引数

1 回の呼び出しで指定される複数の引数(actual_argument_list)は、コンマで区切られたトークン文字列です。コンマが引用符または括弧で囲まれている場合は、引数の区切り文字として認識されません。

マクロ呼び出しにおける引数の数は、マクロ定義のパラメータ数と同数でなければなりません。

各引数の前後の空白は切り捨てられます。

1 つの引数に含まれる連続する空白文字は、1 文字の空白文字に置換されます。

引数は、2 行以上にわたって記述してかまいません。

例 1.12

```
# define ADD(a,b)   a + b
# define MUL(a,b)   (a * b)
マクロ呼び出し      置換されるテキスト
ADD(1,2)            1 + 2
MUL(12,2)           (12 * 2)
ADD(xxx(1,2),3)     xxx(1,2) + 3
ADD(1  + 2,3)       1 + 2 + 3
ADD (  11,3 )       11 + 3
ADD (12345 +
678, 9)             12345 + 678 + 9
```

引数のトークンは、マクロ呼び出しを実行するために解析され、展開する必要がある場合は、マクロ呼び出しを展開する前に展開されます。ただし、引数の先頭に#または##がある場合および後ろに##がある場合は、先に外部呼び出しが展開されます。

例 1.13

```
# define ADD(a,b)        a + b
# define MUL(a,b)        (a * b)
マクロ呼び出し             置換されるテキスト
ADD(MUL(1,2),3)          (1 * 2) + 3
MUL(MUL(ADD(1,2),3),4)   ((1 + 2 * 3) * 4)
```

置換用文字列のパラメータの先頭に#がある場合は、その引数のトークンはマクロ呼び出しのための解析対象にはなりません。

例 1.14

```
# define ONE(a)     #a
# define TWO(a,b)   (a * b)
マクロ呼び出し          置換されるテキスト
ONE (TWO (1,2))     “TWO (1,2)”
```

置換用文字列のパラメータの先頭または後ろに##がある場合、その引数のトークンはマクロ呼び出しのための解析対象にはなりません。

例 1.15

```
# define CAT(a,b)   a ## b
マクロ呼び出し          置換されるテキスト
CAT (CAT(1,2),3)    CAT(1,2)3
```

上記の例の場合、##があるために、外部呼び出し引数は先に展開されません。したがって、外部呼び出しを実行した結果は CAT(1,2)3 になります。置換用テキスト内の識別名 CAT は 2 度現れているため展開されません。

例 1.16

```
# define CAT(a,b)   a ## b
# define TWO(a,b)   (a + b)
マクロ呼び出し          置換されるテキスト
CAT (TWO(1,2),3)    (1 + 2)3
```

上記の例では、識別名 CAT が引数より先に展開されます。このため、結果は (1+2)3 となっています。置換用テキスト内の識別名 TWO は、まだ展開されていないので展開されます。

例 1.17

```
# define CAT(a,b)    a ## b
# define XCAT(a,b)   CAT(a,b)
マクロ呼び出し          置換されるテキスト
XCAT(XCAT(1,2),3)    123
```

上記の例では、XCAT のトークン文字列には##がありませんから、引数が先に展開されます。したがって、内部呼び出し XCAT(1,2)は 12 として展開され、その後に外部呼び出し XCAT（12,3)が 123 として展開されます。

1.5 マクロの削除

構文:

***# undef** identifier*

#undef 指令は、指定した識別名(identifier)の定義をプリプロセッサによって削除します。この後プリプロセッサは、定義を削除した識別名を発見した場合は、再定義されるまでその識別名を無視します。

パラメータを持つマクロとして定義された識別名を削除する方法は、削除する識別名を#undef 指令で指定する以外にはありません。

指定した識別名が#define 指令によって事前に定義されていない場合は、エラーメッセージは表示されません。この場合、指定した識別名は未定義であることを意味しています。

例 1.18

```
# define ONE        1
x = ONE ;
# undef             ONE
y = ONE
# define A(b,c)     b + c
# undef             A
```

上記の例では、変数 x には定数 1 が割り当てられていますが、変数 y には定数の値 1 は割り当てられていません。

1.6 マクロの再定義

マクロを再定義する場合、以下の条件が満たされないとエラーになります。

a) トークン文字列が同じであること。

b) パラメータを持つマクロとして識別名を定義する場合は、パラメータ数が同数であること。

次の再定義はエラーになります。

例 1.19

```
# define A                1+2
# define A                1-2
```

例 1.20

```
# define A                1+2
# define A                1 + 2
```

例 1.21

```
# define A                12
# define A(b)             12
```

例 1.22

```
# define A(b)             b
# define A                b
```

例 1.23

```
# define A(one,two)       one + two
# define A(one)           one + two
```

次の再定義はエラーにはなりません。

例 1.24

```
# define A                1 + 2
# define A                1 + 2
```

例 1.25

```
# define A(one,two)       one + two
# define A(one,two)       one + two
```

例 1.26

```
# define A(one, two)      one + two
# define A(x,y)           x + y
```

1.7 ファイルのインクルード

1.7.1 概要

#include 指令では、プリプロセッサが、コンパイル中に指令を発見した行を、指定されたファイルの全内容に置換します。

ファイルのインクルードを使用すると、**#define** 文および宣言(他のものを含む)の集合体を簡単に取り扱えるようになります。これらは、通常は単独のファイルにまとめて保存されており、コンパイル時に C のソースファイルに読み込まれます。この方法により、さまざまなマクロを集めたいくつものライブラリを、さまざまなソースファイルで使用することが可能です。

指令で指定されたファイルが存在しない場合、フェイタルエラーを表示して、コンパイルを中止します。

インクルードファイルのネスティングは 10 ファイルまでです(ソースファイルを含めて)。

1.7.2 二重引用符を使用するインクルードファイルの指定

構文:

***# include** "filename"*

ファイル名(filename)には、パス指定を含めることができます。ファイル名がパスと共に指定されている場合は、次の順序でファイルの検索が行われます

a) 親ファイルのディレクトリを検索する(親ファイルは、# include 指令を含んでいるファイルのこと)

b) 祖父母ファイルディレクトリ

c) /I コマンドラインオプションを使用して指定されたディレクトリ

d) 環境変数 INCLU8 により設定された標準ディレクトリ

例 1.27

```
# include "\dir1\dir2\abc.c"
```

上記の**#include** 指令では、**#include** 指令を指定した行が\dir1\dir2 ディレクトリにある abc.c ファイルの内容に置換されます。

1.7.3 山形括弧を使用するインクルードファイルの指定

構文:

 # include *<filename>*

ファイル名(filename)にパス指定を含めることができます。

ファイル名の指定にパスが含まれていない場合は、次の順序で検索が行われます。

a) /I コマンドラインオプションを使用して指定されたディレクトリ

b) 環境変数 INCLU8 で設定されている標準ディレクトリ

1.7.4 インクルード指令に含まれるマクロ

構文:

 # include *token_sequence*

この**#include** 指令は、プリプロセッサによるトークン文字列(token_sequence)の展開を行います。

トークン文字列を展開した後、次の 2 形式のどちらかの結果が得られます。

a) 二重引用符で指定されたファイル名

b) 山形括弧で指定されたファイル名

#include 指令の処理は、ファイル名の指定に基づいて行われます。

例 1.28

```
# define FILENAME “file1.c”
# include FILENAME
```

上記の例では、プリプロセッサは file1.c を含めます。

1.8 条件付きコンパイル

1.8.1 概要

条件付きコンパイルでは、次の前処理指令を使用できます。

1. # if
2. # ifdef
3. # ifndef
4. # elif
5. # else
6. # endif

これらの指令は、前処理中にソースファイルのどの部分をコンパイラに送り、どの部分を削除すればいいのかを決定するために定数式または識別子を検査することによって、ソースファイルのコンパイルを部分的に禁止することができます。

1.8.2 条件付きコンパイル指令

構文:

#if *restricted_constant_expression*
[text-block]
*[**#elif** restricted_constant_expression*
[text-block]
.
.
]
[#else
[text-block]
]
#endif

#if 指令に続くテキストブロック(text-block)のテキスト文字列には、特別な制限はありません。2行以上にわたってもかまいません。テキストブロックに前処理指令を持つこともできます。

#elif および**#else** 指令は、任意指定です。**#elif** 指令は、**#if** 指令と**#endif** 指令の間に任意の数だけ記述することができます。**#else** 指令は、**#if** から**#endif** までの間には 1 度だけ記述します。**#else** 指令が使用されている場合は、**#endif** の直前の条件指令を最後の条件指令にします。条件指令ブロックは、**#endif** によって終了させます。

#if とそれに続く**#elif** の間に記述される制限付きの定数式(restricted_constant_expression)は、ゼロ以外の値が見つかるまで評価されます。ゼロの値に続くテキストは、切り捨てられます。ゼロ以外の値に続くテキストは、通常通り処理されます。条件を満たす**#if** または**#elif** が見つかり、テキストが処理されると、その後の**#elif** および**#else** の行とそこに記述されたテキストは切り捨てられます。

すべての式はゼロかどうかを評価している場合、**#else** 指令があれば**#else** に続くテキストが通常通り処理されます。

例 1.29

```
# if 1
      function1 () ;
# endif
```

上記の例では、**#if** 指令に続くテキストが処理されます。

例 1.30

```
# if 0
      function1 () ;
# endif
```

上記の例では、**#if** 指令に続くテキストは、式の結果がゼロであるために切り捨てられます。

例 1.31

```
# if 1
      function1 () ;
# elif 0
      function2 () ;
# endif
```

上記の例では、式の結果がゼロ以外のため、**#if** 指令に続くテキストが処理されます。**#elif** 指令に続く定数式の評価は行われません。**#elif** 指令に続くテキストは切り捨てられます。

例 1.32

```
# if 0
      function1 () ;
# elif 1
      function2 () ;
# endif
```

上記の例では、**#if** に続くテキストは処理されません。**#elif** 指令に続く定数式の評価は行われます。式の結果がゼロ以外の場合には、**#elif** 指令に続くテキストが処理されます。

例 1.33

```
# if 0
      function1 () ;
# elif 0
      function2 () ;
# endif
```

上記の例では、**#if** 指令と**#elif** 指令は、どちらの式もゼロを評価しているため、この 2 つの指令に続くテキストは切り捨てられます。

1.8.3 制限付き定数式

前処理指令の中にある定数式には、特定の制限があります。定数式は、整数の式でなければなりません。sizeof 式、列挙定数、浮動小数点定数、キャスト式を含めることはできません。

マクロが存在する場合は展開されます。マクロを展開した後に残る識別名はすべて 0L に置換されます。

以下は、#if と#elif 指令での制限付き定数式の使用例です。

例 1.34

```
# if 1 +2
```

例 1.35

```
# if 1 + 2 * 3 % 4
```

例 1.36

```
# if A
```

例 1.37

```
# if (1 + 2) / 5
```

例 1.38

```
# if (1 << 2) == A
```

例 1.39

```
# if A || B & C
```

例 1.40

```
# if A ? B : C
```

以下は、誤りの例です。

例 1.41

```
# if A = 2
```

例 1.42

```
# if X += 5
```

例 1.43

```
# if X ++
```

例 1.44

```
# if &X
```

例 1.45

```
# if sizeof (struct A)
```

例 1.46

```
# if A, C
```

例 1.47

```
# if 1.2
```

1.8.4 defined 演算子

構文:

defined identifier

defined (identifier)

上の構文を持つすべての式は、識別名(identifier)がプリプロセッサに定義されている場合は 1L で、定義されていない場合は 0L で置換されます。

例 1.48

```
# define A 1
# if defined (A)
       printf (“この部分はコンパイルされます”) ;
# endif
# if defined (B)
       printf (“この部分はコンパイルされません”) ;
#endif
```

defined 演算子は、他の演算子と一緒に現れることもあります。

例 1.49

```
# define A 1
# if ! defined (A)
       printf (“この部分はコンパイルされません”) ;
# endif

# if defined (A) - 1
       printf (“この部分はコンパイルされません”) ;
# endif
```

1.8.5 ネスティング

#if、**#elif**、**#else**、**#endif** 指令は、別の**#if** 指令のテキスト部分でネストすることができます。**#elif**、**#else** および**#endif** 指令は、それぞれ直近の先行する**#if** 指令に従属します。

例 1.50

```
# if 0
       # if 1
              printf (“この部分はコンパイルされません”) ;
       # endif
# endif
# if 1
       # if 0
              printf (“この部分はコンパイルされません”) ;
       # endif
       # if 1
              printf (“この部分はコンパイルされます”) ;
       # endif
# endif
```

指定できるネスティングは、32 階層までです。

1.8.6 #ifdef と#ifndef によるシンボル定義のテスト

構文:

 # ifdef *identifier*

 # ifndef *identifier*

#ifdef指令と**#ifndef**指令は、**#if**指令の記述可能な場所であればどこに記述してもよい指令です。**#ifdef** 指令に続くテキストは、指定されている識別名(identifier)がマクロの場合にコンパイルされます。**#ifndef** 指令に続くテキストは、指定されている識別名がマクロではない場合にコンパイルされます。

例 1.51

```
# define A 1
# ifdef A
      printf (“この部分はコンパイルされます”) ;
# endif

# ifdef B
      printf (“この部分はコンパイルされません”) ;
# endif

# define B 2
# ifndef B
      printf (“この部分はコンパイルされません”) ;
# endif
# undef A
# ifndef A
      printf (“この部分はコンパイルされます”) ;
# endif
```

1.9 LINE

構文:

 # line *constant [“filename”]*

#line 指令では、プリプロセッサが以下の変更を行います。

a) 次のソース行番号を定数(constant)で指定された行番号へ

b) 現在のソースファイルの名前を指定されたファイル名(filename)へ

定数の値は、整数でなければなりません。範囲は 1～32,767。1 および 32,767 を含みます。

ファイル名の指定は任意です。ファイル名は、二重引用符で囲んで文字列リテラルであることを示します。

 # line 指令のマクロは、翻訳する前に展開されます。

行番号とファイル名は、コンパイラがコンパイル中に生成するエラーメッセージで使用されます。

例 1.52

```
# line 124
```

次のソース行の行番号が「124」に変わります。ソースファイル名の変更は行われません。

例 1.53

```
# line 1234 “file.c”
```

次のソース行の行番号が「1234」に変わります。ソースファイル名は「file.c」に変わります。

例 1.54

```
# define LINENUMBER 1234
# define FILENAME “file1.c”
# line LINENUMBER FILENAME
```

次のソース行の行番号が「1234」に変わります。ソースファイル名は「file1.c」に変わります。

1.10 ERROR

構文:

*# **error** [token_sequence]*

 # error 指令は、プリプロセッサによる診断エラーメッセージを出力させます。エラーメッセージには、任意のトークン文字列(token_sequence)を含められます。

コンパイラの表示するエラーメッセージには、エラー番号、ソースファイル名、ソース行の行番号が含まれます。

トークン文字列の中のマクロは展開されません。

例 1.55

```
# error this is an old version
```

上記の**# error** 指令は、コンパイラに「**# error**：this is an old version」というメッセージを表示させます。

例 1.56

```
# define ERROR__MESSAGE this is the error message
# error ERROR__MESSAGE
```

上記の**# error** 指令は、コンパイラに「ERROR_MESSAGE」というメッセージを表示させます。マクロは展開されません。

1.11 ミックスドランゲージプログラミング

構文:

 # asm

[assembly text]

 # endasm

#asm 指令と**#endasm** 指令は、ミックスドランゲージプログラミングの使用時の負担を軽減します。**#asm** と**#endasm** の間で指定するアセンブリテキスト(assembly text)は、処理されません。

アセンブリテキストは、複数行にわたることが可能です。

#asm 指令は、アセンブリテキストの先頭を示します。**#endasm** 指令は、アセンブリテキストの終了を示します。

例 1.57

```
# asm
l        er0,NEAR _b            ;; a = b + c
l        er2,NEAR _c
add      er0,er2
st       er0,NEAR _a
# endasm
```

1.12 事前定義マクロ

事前定義のマクロは、次のとおりです。

1. __LINE__
2. __FILE__
3. __DATE__
4. __TIME__
5. __STDC__
6. __CCU8__
7. __VERSION__
8. __ARCHITECTURE__
9. __DEBUG__
10. __MS__
11. __ML__
12. __NEAR__
13. __FAR__
14. __UNSIGNEDCHAR __
15. __NOROMWIN__

上記の事前定義マクロは、再定義または未定義することはできません。

1. __LINE__

   __LINE__は、10 進定数を展開します。10 進定数は、コンパイルしたソースの現在の行番号です。

2. __FILE__

   __FILE__は、文字列リテラルを展開します。コンパイルしたファイル名です。

3. __DATE__

   __DATE__は、文字列リテラルを展開します。次の形式で記述されるコンパイル実施日です。

   “Mmm dd yyyy”

4. __TIME__

   __TIME__は、文字列リテラルを展開します。次の形式で記述されるコンパイル時刻です。

   “hh:mm:ss”

5. __STDC__

   __STDC__は、10 進定数 0 を展開します。この定数値は、ANSI 規格に準拠するコンパイルの場合は 1 になります。

6. __CCU8__

   __CCU8__は、10 進定数 1 を展開します。

7. __VERSION__

   __VERSION__は、文字列リテラルを展開します。次の形式で記述される現在のバージョン番号です。

   “Ver.X.YY”

   「X.YY」が現在のバージョン番号です。

8. __ARCHITECTURE__

   __ARCHITECTURE__ は、文字列リテラルを展開します。/T オプションで指定すると、次の形式になります。

   “core”

   core は、/T オプションで指定された文字列です。/T オプションを省略すると、置き換わるテキストは“”です。

9. __DEBUG__

   __DEBUG__は、C のソースプログラムを、/SD オプションを指定してコンパイルした場合に、10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

10. __MS__

    __MS__は、C のソースプログラムを、/MS オプションまたはデフォルトメモリモデルオプションを指定してコンパイルした場合に、10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

11. __ML__

    __ML__は、C のソースプログラムを、/ML オプションを指定してコンパイルした場合に 10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

12. __NEAR__

    __NEAR__は、C のソースプログラムを、/near オプションまたはデフォルトデータアクセス識別子オプションを指定してコンパイルした場合、または、マクロが NEAR プラグマの後に指定されている場合に、10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

13. __FAR__

    __FAR__は、C のソースプログラムを、/FAR オプションを指定してコンパイルした場合、または、マクロが FAR プラグマの後に指定されている場合に、10 進定数 1 を展開します。オプションが指定されていないコンパイルでは、このマクロは定義されません。

14. __UNSIGNEDCHAR__

    __UNSIGNEDCHAR__は、C のソースプログラムを、/J オプションを指定してコンパイルした場合に、10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

15. __NOROMWIN__

    __NOROMWIN__ は、C のソースプログラムを、/NOWIN オプションを指定してコンパイルした場合に、10 進定数 1 を展開します。そうでない場合には、このマクロは定義されません。

次の条件を与えると、事前定義マクロは、以下のように展開されます。ファイル名は test.c、コンパイルしている行の番号は 200、コンパイル実施日は 2000 年 9 月 20 日、コンパイル時刻は 10 時 20 分 30 秒。

例 1.58

マクロ呼び出し	置換されるテキスト
__LINE__	200
__FILE__	“test.c”
__DATE__	“Sep 20 2000”
__TIME__	“10:20:30”
__STDC__	0

例 1.59

```
int i ;
void
func (void)
{
	i = __MS__ ;
}
```

上記の C ソースプログラムを、/MS オプションまたはデフォルトメモリモデルオプションでコンパイルした場合の結果は、次のとおりです。

```
int i ;
void
func (void)
{
	i = 1 ;
}
```

2. 字句規約

2.1 文字セット

ここでは、CCU8 での字句規約について説明します。前処理の後、ソースプログラムは、この字句規約に基づいて、一連のトークンに分割されます。

C 言語の文字セットは、文字、数字、C 言語で特定の意味を持つ句読点から構成されています。C プログラムは、C 言語の文字セットの文字を組み合わせて作成する意味を持ついくつもの文で構成されます。

次の文字は、C 言語では、定数、識別子、キーワードを構成する場合に使用します。

英字			(A-Z, a-z)						
数字			(0 - 9)						
!	#	‘	“	%	&	(	)	=	~
-	^	\	|	,	.	/	?	{	}
<	>	;	:	+	*	[	]	_	
空白(20H)			タブ(09H)			改行(0DH)			
行送り(0AH)			改ページ(0CH)			垂直タブ(0BH)			

CCU8 は、大文字と小文字を別の文字として扱います。

ブランク(空白)、水平タブ、垂直タブ、改行(NL)、行送り、改行(CR)、および改ページをまとめて空白類文字と総称します。これらの文字は、コンパイラでは、トークンを区切る場合以外は無視されます。ユーザがプログラム中に定義した定数や識別子などの項目同士を分離させる文字です。

2.2 トークン

コンパイラが認識する C ソースプログラム内の基本要素は、文字集合で、これを「トークン」と呼びます。トークンは、ソースプログラムのテキストで、コンパイラは、構成要素の解析を行いません。CCU8 は、次のトークンを識別します。

・識別子

・キーワード

・コメント

・定数

・演算子

2.2.1 識別子

識別子は、英字、数字および下線の連続する列です。最初の文字には、文字または下線を使用します。CCU8 のデフォルト設定は、31 文字以内の識別子を指定できます。この文字数を超える識別子が指定されている場合、CCU8 は、ワーニングメッセージを表示し、最初の 31 文字だけを 1 つの識別子として使用します。ただし、コマンドラインオプションの/SL を使用すると、識別子の最大長を指定することができます。許容範囲は 31～254 文字で、両端を含みます。

次に、識別子の例を示します。

例 2.1

```
i
count
number
end_of_file
Minus
SUBTRACT_THIS
_var
```

2.2.2 キーワード

コンパイラがキーワードとして確保している識別子です。定義を変えることはできません。CCU8 では、データの種類、記憶クラス、文の識別に使用しています。キーワードは、必ず小文字で記述します。CCU8 のキーワードは次のとおりです。

auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto

if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

CCU8 は、ANSI C 標準の拡張である、次のキーワードをサポートしています。ただし、**/Za** オプションを指定すると、これらのキーワードは識別子として扱われます。

__DI	__EI	__asm	__far	__near	__noreg

2.2.3 コメント

コメントは、「/*」と「*/」で区切られた部分で、空白文字の許可される場所であれば、どこにでも記述することができます。コメントのテキストには、コメント終了文字「*/」を除くすべての文字を使用できます。コメントをネストすることはできません。文字列または文字列リテラルの中に記述することはできません。

例 2.2

```
i. /*これはコメントです。*/
ii. /*コメントの/* ネスティングは*/ 禁止されています。*/
```

2 行目(ii)に対して、エラーが出力されます。

CCU8 コンパイラは、コメントの拡張をサポートしています。シンボル「//」は、物理的ソース行の任意の場所で使用することができます(文字定数または文字列リテラルを除く)。//から行終了までの中にあるすべての文字が、コメント文字として扱われます。コメントの終わりは、行終了で示します。

例 2.3

```
// すべてのファイルをクローズする
for (lvar = 0; lvar < lcount; lvar++ )
{
     a++ ;                  // a をインクリメントする
}
```

2.2.4 定数

C 言語の定数は、プログラムで変更できない固定された値、文字、文字列をいいます。CCU8 は、整数、浮動小数点、文字、文字列の 4 種類の定数をサポートしています。

2.2.4.1 整定数

整定数は、16 進数、10 進数または 8 進数で表現します。10 進整定数の 1 番目の文字は、数字である必要があります。0X または 0x で始まる数字の並びは、16 進整定数として扱われます。数字の並びが 0 で始まる場合は、8 進整定数で、それ以外は、10 進整定数です。

	有効文字	**接頭語**
16 進数	0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f	0X または 0x
10 進数	0 1 2 3 4 5 6 7 8 9	なし
8 進数	0 1 2 3 4 5 6 7	0

整定数を「符号なし」と指定するには、「u」または「U」を接尾語として使用します。long 型として指定する場合には、「l」または「L」を接尾語として使用します。

どの整定数にも、値に応じて型が与えられます。定数が式で使われる場合、定数型に応じて変換がなされます。変換規則をまとめると次のようになります。

* 整定数の型は形式、値および接尾語によって決定します。整定数の型については、整数定数の種類を示す表の先頭にあります。

接尾語のない 10 進数 : **int** 型、**long int** 型、**unsigned longint** 型

接尾語のない 8 進数または 16 進数 : **int** 型、**unsigned int** 型、**long int** 型、**unsigned long int** 型

接尾語 **u** または **U** を使用 : **unsigned int** 型、**unsigned long int** 型

接尾語 **l** または **L** を使用 : **long int** 型、**unsigned long int** 型

接尾語 **u** または **U** と **l** または **L** の 2 つを使用 : **unsigned long int** 型

次の表は、CCU8 の 8 進定数および 16 進定数の範囲と対応する型を示しています。

16 進数の範囲	**8 進数の範囲**	**型**
0x0～0x7fff	0～077777	int 型
0x8000～0xffff	0100000～0177777	unsigned int 型
0x10000～0x7fffffff	0200000～017777777777	long 型
0x80000000 to 0xffffffff	020000000000 to 037777777777	unsigned long 型

整定数は、文字「l」または「L」を付加すると、強制的に long 型になります。

次に、いくつかの整定数の例を示します。

例 2.4

```
0x177AF       /*16 進数の整数*/
0167          /* 8 進数の整数*/
1826          /*10 進数の整数*/
0X1abe        /*16 進数の整数*/
10l           /*10 進数の long 型整数*/
0xabL         /*16 進数の long 型整数*/
0333l         /*8 進数の long 型整数*/
```

2.2.4.2 浮動小数点定数

浮動小数点定数は、整数部(10 進)と小数部(e または E の文字)で構成され、任意指定の符号付き整数指数部も使用できます。10 進数で構成される整数部および小数部は、どちらかを省略することができます。小数点に続く数字と指数部のどちらかを省略することはできますが、両方を省略することはできません。

浮動小数点定数の型には、**float** 型と **double** 型があります。型の決定は、接尾語です。F は float 型、L または l は **long double** 型を示し、それ以外は **double** 型とみなされます。**Long double** 型定数の扱いは、**double** 型定数とほぼ同じです。以下に浮動小数点定数の例を示します。

例 2.5

```
1.0e10f
.75
1.03e-12L
3.0
120e22
10e04
-0.0021
```

2.2.4.3 文字定数

文字定数は、引用符「‘’」で囲まれた 1 文字の ASCII 文字です。文字定数として使用できるのは 1 バイト文字だけです。エスケープシーケンスは、1 文字として扱われますから、文字定数として使用することできます。引用符とバックスラッシュを文字定数として使用する場合は、接頭語としてバックスラッシュを付ける必要があります。

例 2.6

‘ ’ 1 つの空白
‘z’ 小文字の z
‘\n’改行文字
‘\\’ バックスラッシュ
‘\’’ 引用符

2.2.4.4 文字列リテラル

文字列リテラルは、二重引用符「“..”」で囲まれた文字の並びです。文字列は、「文字型の配列」型で、記憶クラスは **static**、そして、与えられた文字で初期化されます。

隣接する文字列は、1 つの文字列として連結されます。連結後、文字列にヌルバイト「**\0**」が付加され、この文字列を検査するプログラムが文字列の終了を発見できるようにします。連結されていない文字列も同様で、すべての文字列には、終了位置を示すためのヌルバイトが付加されます。文字列リテラルは、エスケープシーケンスを含めることができます。

2 行以上にわたる文字列リテラルを構成するには、バックスラッシュを入力し、Return キーを押します。これによって、バックスラッシュの直後に続く改行文字をコンパイラに無視するように知らせることができます。以下に例を示します。

“この 2 行にわたる文字列は、\
1 行の文字列として連結されます”

上記の表記結果は、次の表記と同じです。

“この 2 行にわたる文字列は、1 行の文字列として連結されます”

空白文字だけで区切られた複数行の文字列は、1 つの文字列として連結されます。次の文字列を例にして、説明します。

“1 行目、”
“2 行目”

上記の文字列は、次のように連結されます。

“1 行目、2 行目”

エスケープシーケンスは、文字列リテラルとして使用できます。文字列リテラル内に二重引用符またはバックスラッシュを使用するには、エスケープシーケンスを使用します。

例 2.7

i. “One\\two”

ii. “\”Do it\” Mike said.”

2.2.4.5 エスケープシーケンス

文字列リテラルまたは文字定数は、「エスケープシーケンス」を含めることができます。エスケープシーケンスは、空白文字と非図形文字を表現する文字の組み合わせです。エスケープシーケンスは、バックスラッシュ「\」とその後に続く 1 文字または 1 組の数字で構成します。

エスケープシーケンスは、通常は、端末やプリンタでの復帰(CR)やタブの動作を指定するために使用され、印字できない文字や、二重引用符「“」のような特殊な意味を持つ文字を定数に表現します。CCU8 のエスケープシーケンスを次の表に示します。

エスケープシーケンス	名前	
\n	改行	NL (LF)
\t	水平タブ	HT
\v	垂直タブ	VT
\b	後退	BS
\r	復帰	CR
\f	改ページ	FF
\a	ベル(警告音)	BEL
\’	一重引用符	
\”	二重引用符	
\\	バックスラッシュ	
\ooo	8 進形式の ASCII 文字	
\xhh	16 進形式の ASCII 文字	

上記の表にない文字の前にバックスラッシュがある場合は、このバックスラッシュは無視され、文字定数に表現されます。たとえば、「\m」という文字パターンは、文字列リテラルおよび文字定数では「m」を意味します。

「\ooo」という列は、プログラマが ASCII 文字セットの任意の、文字を 3 桁の 8 進文字コードで指定するために使用します。16 進数のエスケープシーケンスの中で、バックスラッシュ「\」または文字「x」の後ろに続く 16 進数の数字は、整数の文字定数を表現する 1 つの文字の構成の一部とみなされます。16 進定数の数値は、このように構成し、必要な文字の値を示します。16 進数の各エスケープシーケンスは、エスケープシーケンスを構成する文字列の中で最も項目が多いです。たとえば、ASCII 水平タブ文字は、通常の C 言語のエスケープシーケンス「\t」で指定することができるほか、「\011」(8 進数)または「\x09」(16 進数)のコードでも指定できます。

8 進数および 16 進数のエスケープシーケンスを構成するには、必ず 1 桁の数字を指定します。たとえば、「\11」「\011」「\x9」「\x09」は、有効なエスケープシーケンスです。

2.2.5 演算子

演算子は、値をどのように扱うかを指定するシンボルです。それぞれのシンボルは、トークンと呼ばれる単一のユニットと解釈されます。次の表は、C 言語の単項演算子、2 項演算子、3 項演算子の一覧です。

単項演算子

!	~	-	*	&	+	++	--	sizeof

2 項演算子

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

3 項演算子

?:

「*」「**&**」「**-**」「**+**」の 4 つの演算子は、上記の表では、単項演算子と 2 項演算子の両方に列挙されています。どちらの演算子として扱うかは、プログラムでシンボルが使用されている部分の流れに基づいて決定されます。

3. プログラムの構造

3.1 ソースプログラム

ここでは、CCU8 で実装する C 言語のマニュアルで使用している用語を定義した後、C プログラムの構造を説明します。

C のソースプログラムは、任意の数の指令、宣言、定義および文の集合体です。各構成要素については、この後で、簡単に説明します。構成要素のプログラム中に現れる順序に関する制約はありません。

指令

指令は、コンパイルを実行する前に C プリプロセッサが実行する特定の動作をプログラムテキスト上で指定します。指令の詳細な説明は、「第 1 章 プリプロセッサ」を参照してください。

宣言と定義

宣言は、変数、関数または型の名前と属性の関係を確立します。C 言語では、すべての変数は、使用する前に必ず宣言しなければなりません。

変数定義は、宣言と同じ関連付けを行いますが、変数に割り当てられたものの格納も行われます。すべての定義は宣言ですが、宣言のすべてが定義ではありません。

例 3.1

```
const int a = 10 ;                  /* 変数定義*/
int b ;                             /* (外部)*/
extern int function (int, char) ; /* 関数宣言または関数プロトタイプ*/
extern long c ;                     /* 外部変数宣言*/
extern float f ;

main ()
{
       int local1 ;                 /* 変数定義*/
       char local2 ;                /* (内部) */
       local1 = local2 ;            /*実行文*/
       c = b + a + f ;
}
```

3.2 ソースファイル

1 つのソースプログラムは、複数のソースファイルに分割された状態で持つことができます。C 言語のソースファイルは、C プログラム全体または一部を持つテキストファイルです。コンパイル中、個々のソースファイルは個別にコンパイルします。

ソースファイルには、任意に組み合わせた指令、宣言、定義が含まれます。関数定義や大きなデータ構造を複数のソースファイルに分割することはできません。ソースファイルの最後の文字は、必ず改行文字またはファイル終了文字にします。

ソースファイルは、必ずしも実行文を含まなくてもかまいません。たとえば、変数定義を 1 つのソースファイルにまとめ、変数を使用する別のソースファイルに、変数への参照を宣言することができます。この方法をとると、定義を見つけるのも変更するのも簡単になります。同じ理由から、マクロと**#define** 文を、必要なときだけソースファイルで参照できる別個のインクルードファイルにまとめる方法は、高い頻度で使用されています。

あるソースファイルの中の指令は、そのソースファイルとインクルードファイルにだけ適用されます。さらに、各指令は、そのソースファイル中、指令の後の部分にだけに効力を持ちます。ソースプログラム全体に共通の指令を適用するには、対象となる指令がプログラムを構成するすべてのソースファイルに含まれていなければなりません。

3.3 関数とプログラムの実行

すべての C プログラムは、「main」と名付けなければならない基本(main)関数を持ちます。main 関数は、プログラムの実行開始位置を示します。通常は、プログラム中の関数の呼び出しを行って、プログラムの実行を制御します。また、通常、プログラムの実行は、main 関数の終了する位置で終了しますが、実行環境に基づくさまざまな理由から、他の場所で終了させることもできます。

ソースプログラムは、通常、それぞれが 1 つまたは複数の特定のタスクを実行するように設計された複数の関数を持ちます。main 関数は、このような関数を呼び出して、1 つまたは複数の特定のタスクを実行します。main 関数は、別の関数の呼び出しを行うと、呼び出した関数の 1 行目の文から実行するために、実行の制御を呼び出した関数に移します。制御を受け取った関数は、return 文を実行したとき、または、自分自身の終わりに達したときに、main 関数に制御を返します。

関数宣言を行うと、関数はパラメータを持つことができます。パラメータを持つ関数が別の関数を呼び出すとき、呼び出される関数は、呼び出す関数から値を受け取ります。この値を引数と呼びます。

関数間の引数の受け渡しは、「値渡し」という方法で行われます。

3.4 寿命と可視性

変数および関数をプログラムでどのように使用できるのかを限定する規則を理解するための不可欠なコンセプトは 3 つあります。ブロック(または複文)、寿命(エクステントということも)、可視性(スコープということも)です。

3.4.1 ブロック

ブロックは、宣言、定義および文を 1 つの中括弧で囲んだ 1 つの列です。C プログラムには、2 種類のブロックがあります。そのうちの 1 つは、複文です。もう 1 つは、関数の本体を構成する 1 つの複文と関数の先頭(関数名、戻り値の種類、仮パラメータ)で構成される関数定義です。1 つのブロックは、複数のブロックを包含することができます。ただし、ブロックに関数定義を持つことができないという制限があります。あるブロックが他のブロックの中に入ることを「取り込む側のブロックにネストされる」といいます。

すべての複文は、中括弧で囲みます。しかし、中括弧に囲まれたものがすべて 1 つの複文の構成要素というわけではありません。たとえば、中括弧の中に配列指定または構成要素が存在することがありますが、どちらも複文として扱われません。

3.4.2 寿命

寿命は、変数または関数を含むプログラムが有効な期間のことです。あるプログラム中のすべての関数は、プログラム実行時には必ず存在しています。

変数の寿命には、グローバルまたはローカルの 2 種類があります。寿命がグローバルの場合(グローバル項目)、プログラムの存続期間中を通して有効な格納先と定義済の値を持ちます。ローカル項目は、自身が定義または宣言されたブロックの中でだけ有効な格納先と定義済の値を持ちます。プログラムの制御がローカル項目のブロックに移動してくるたびに、ローカル項目には、新しい格納先が割り当てられ、プログラムがブロックから抜けるときに確保された格納先が(当然、値も)消失します。

3.4.3 可視性

可視性は、項目を名前で参照できるプログラム中のある部分を決定します。1 つの項目は、プログラムの有効範囲に包括された部分でだけ見えるようになります。有効範囲は、ファイル、関数、ブロックまたは関数プロトタイプに限定されているといえます。

3.5 命名クラス

C プログラムでは、種類の異なるさまざまな項目を参照するときに識別子を使用します。識別子は、関数、変数、仮パラメータ、共用体メンバーおよびプログラムが使用する他の項目に与えられていなければならないものです。ここで説明している規則が守られていれば、複数のプログラムが 1 つの識別子を共用できる C 言語の機能を利用することができます。

コンパイラは、種類の異なる項目が使用する識別子を区別するために、命名クラスを設定します。識別子の名前は各クラスで一意にして混乱をさけるようにしますが、複数の命名クラスでは、複数の同一名が存在する可能性があります。これは、つまり、指定された複数の項目の命名クラスが異なっていれば、この指定された複数の項目は 1 つの識別子を使用できることを意味しています。コンパイラは、プログラムの前後関係から、参照先を見つけ出します。

この後、C プログラムで指定できる項目の種類と名前を付けるための規則を説明します。

ラベル付き文

ラベル付き文は、1 つの独立した命名クラスを構成します。各ラベル付き文は、1 つの関数内では一意にします。しかし、他の関数のラベル付き文と区別する必要はありません。

変数と関数

変数名と関数名は、仮パラメータ名と typedef 名と一緒に 1 つの命名クラスの中にあります。このため、変数名と関数名は、このクラスの中で同じ可視性をもつ他のものの名前と異なる名前にしなければなりません。しかし、変数名と関数名は、関数ブロックの中で再定義することができます。

仮パラメータ

1 つの関数に対する仮パラメータの名前は、ローカル変数の名前と一緒にグループ化されます。このため、仮パラメータ名はローカル変数名と区別がつくようにしなければなりません。仮パラメータは、関数の開始レベルでは再定義できませんが、関数の本体部でネストされたブロックの中では再定義できます。

typedef 名

「**typedef**」キーワードで定義した型名は、変数名および関数名と一緒に 1 つの命名クラスの中にあります。このため、**typedef** 名は、同じ可視性を持つすべての変数と関数と仮パラメータとは、区別できる名前でなければなりません。変数名と同様に、**typedef** の型に使用する名前は、プログラムのブロックの中で再定義できます。

タグ

構造体、共用体および列挙タグは、1 つの命名クラスの中でグループ化されます。これらのタグは、同じ可視性を持つ他のタグと区別の付く名前にする必要があります。タグは、他の名前と競合しません。

メンバー

各構造体および各共用体のメンバーは、1 つの命名クラスを構成します。このため、各メンバーの名前は、この構造体または共用体の中では一意にしなければなりませんが、プログラム中において他のものの名前と区別する必要はありません。他の構造体および共用体のメンバーと区別する必要もありません。

例 3.2

```
struct name {
                char * name ;
                int type ;
                int scope ;
            } name ;
```

構造体タグ、構造体メンバーおよび変数名は 3 つの異なる命名クラスにあるので、この例で、「name」という名前を持つこの 3 つの項目の名前は競合しないで、それぞれ区別されます。

3.6 データ型

CCU8 は、いくつかの基本データ型と派生データ型をサポートします。

基本型

CCU8 は、基本データ型を何種類かサポートしています。それらは **char** 型、**int** 型、**long** 型、**float** 型および **double** 型です。**short int** 型、**int** 型および **long int** 型の 3 種類の大きさの整数を扱うことができます。**char** 型および **int** 型は、signed あるいは unsigned のオブジェクトとして宣言することができます。各型のサイズおよび最小値/最大値については、第 4.2 章を参照してください。

上記の基本型オブジェクトは、すべて数値として解釈できます。このため、算術式とも呼ばれます。

すべてのサイズの **char** 型および **int** 型は、符号の有無にかかわらず、まとめて整数型と呼ばれます。

float 型、**double** 型および **long double** 型は、浮動少数点型と呼ばれます。

派生型

基本型の他に、次の方法で基本型から構成された派生型が概念的には無限に存在します。

指定した型のオブジェクトの配列

指定した型のオブジェクトを返す関数

指定した型のオブジェクトへのポインタ

さまざまな型の一連のオブジェクトを含む構造体

さまざまな型の複数のオブジェクトのうちの 1 つを格納する共用体

4. 宣言

4.1 概要

宣言は、それぞれの識別子に与える解釈を指定します。宣言用として識別子に関連付けた記憶領域を確保する必要はありません。固有の記憶領域が確保されている宣言を定義と呼びます。宣言の形式は次のとおりです。

declaration : [declaration_specifiers] [init_declarator_list] ;

declaration : __asm (string)

C 言語のすべての変数は、使用する前に明示的に宣言する必要があります。

宣言子は、角括弧([])、アスタリスク (*)または括弧(( ))などによって性質の変わる識別子です。宣言指定子は、型と記憶クラスの指定子の列で構成されます。

declaration_specifiers :

storage_class_specifier [declaration_specifiers]

type_specifiers [declaration_specifiers]

type_qualifiers [declaration_specifiers]

init_declarator_list :

init_declarator

init_declarator_list , init_declarator

init_declarator :

declarator

declarator = initializer

4.2 型指定子

CCU8 では、次の型指定子をサポートしています。

void	char	int	short	enum
long	float	double	signed	
unsigned	struct	union	typedef	

キーワードの **signed** と **unsigned** は、整数のすべての型の接頭語として使用できるだけでなく、それぞれ、単独で記述する場合には、**signed int** 型および **unsigned int** 型とみなされます。

int を単独でキーワードとして使用する場合は、**signed int** 型として解釈されます。キーワードの **long** と **short** は、単独で使用される場合は **long int 型および short int** 型として解釈されます。デフォルト設定では、**char** だけが指定されている場合は、**signed char** 型と解釈されます。しかし、コマンドラインで**/J** オプションが指定されている場合は、コンパイラは、デフォルト設定の **char** 型を **unsigned char** 型として解釈します。

signed char 型、**signed int** 型、**signed short int** 型および **signed long int** 型は、unsigned と対で整数型と呼ばれます。**float** 型および **double** 型の型指定子は、浮動少数点型として扱われます。これらの整数型および浮動少数点型の型指定子は、変数宣言または関数宣言で使用できます。

キーワード **void** の 3 通りの使用方法は、次のとおりです。

1. **void** は、戻り値のない関数の宣言に使用します。
2. 型指定のないポインタの宣言に使用します。
3. **void** が関数名の後の括弧内で単独である場合、その関数には引数がないことを示しています。

基本型の値の記憶領域と範囲は、次のとおりです。

型	記憶領域	値の範囲
char	1 バイト	-128～127
unsigned char	1 バイト	0～255
short,int	2 バイト	-32,768～32,767
unsigned short,unsigned int	2 バイト	0～65,535
long	4 バイト	-2,147,483,648～2,147,483,647
unsigned long	4 バイト	0～4,294,967,295
float	4 バイト	IEEE 標準形式 1.17549435e-38～3.40282347e+38
double	8 バイト	IEEE 標準形式 2.2250738585072014e-308～ 1.7976931348623157e+308

long double 型の指定子も使用できます。意味は、**double** 型と同じです。

4.3 型修飾子

1. **const**
2. **volatile**

宣言するオブジェクトの特殊な特性を示すために型を修飾できます。CCU8 のサポートする型修飾子は、**const** と **volatile** です。

const 型修飾子は、オブジェクトが変更不可能であることを宣言するために使用されます。キーワードの **const** は、あらゆる基本型または集合型の修飾子として使用できます。**typedef** は、**const** 型修飾子によって修飾されます。キーワード **const** を集合型宣言子の修飾子として含む宣言は、集合型の各要素は、変更不可能であることを示します。項目の宣言に **const 型**の修飾子だけしかない場合は、この項目は **constint** 型として扱われます。

CCU8 は、このような変数を ROM(Read Only Memory)に割り当てます。**const** 型修飾子が使用できるのは、グローバル変数だけです。コマンドラインで**/NOWIN** オプションが指定されなかった場合、CCU8 は **const** で修飾された near 変数を、ROMWINDOW 領域に割り当てます。コマンドラインで**/NOWIN** オプションが指定された場合は、**const** で修飾された near 変数を ROMWINDOW 領域に割り当てることができないため、CCU8 はこの変数に対してワーニングメッセージを出力します。

CCU8 は、ワーニングメッセージを出力した後、ローカル自動変数および関数の **const** 修飾子を無視します。**/Za** オプションが指定されている場合は、関数パラメータは **const** で修飾され、このパラメータが変更された場合は、エラーを出力します。

構造体の各メンバーを **const** で修飾することはできません。ただし、**/Za** オプションを指定した場合、メンバーは **const** で修飾でき、メンバーが変更されるとエラーが出力されます。

構造体および共用体の場合、タグを **const** で修飾することはできません。**const** で修飾できるのは、**構造体/共用体**の変数だけです。CCU8 は、**構造体**および**共用体**のタグでは、**const** を無視します。構造体/共用体のタグと共に指定されている **const** 修飾子は、**構造体/共用体**のタグ宣言と共に指定されている変数があれば、その変数の修飾子として扱われます。

例 4.1

```
const struct tag {
                        int a ;
                        char b ;
                    } var0 ;
struct tag var1 ;
const struct tag var2 ;
```

上記の例では、**構造体**のタグ「tag」の宣言で const が使用されていますが、この const は無視されます。このため、読み取り専用メモリに置くためには、この「tag」を使用して宣言した変数を、**const** で修飾する必要があります。ただし、**構造体**のタグ「tag」の宣言で定義された変数は、**const** で修飾されます。つまり、上記の例では、「var1」は **const** で修飾されず、「var0」と「var2」は **const** で修飾されます。

構造体の個々のメンバーは、const で修飾することはできません。

例 4.2

```
struct tag1 {
                    int a ;
                    const char b ;
                } var1 ;
struct tag1 var2 ;
```

上記の例では、構造体メンバー「b」の宣言に const が使用されていますが、ワーニングの出力後、この const は無視されます。

typedef 識別子は、**const** に修飾されます。

例 4.3

```
typedef const int ca ;
ca const_identifier ;
```

上記の例では、typedefed 識別子は **const** に修飾されます。このため、typedef で修飾された変数「ca」を使用して宣言された変数の「const_identifier」も **const** に修飾されます。

volatile 型修飾子は、自身が現れるプログラムの制御を超えた何かによって合法的に変更される値を持つアイテムを宣言します。

キーワードの **volatile** は、const と同じ環境で使用できます。項目は、const または volatile のどちらにもなり得ます。

volatile で修飾された項目は、この項目が使用される式の最適化を抑制します。

例 4.4

```
volatile int input ;
```

```
volatile char * key_ptr ;
```

上記の例では、「input」の値は、プログラムの制御範囲を超えたところで変わります。同様に、「key_ptr」によって示された位置にある内容も、プログラムの制御範囲を超えたところで変わります。

4.4 宣言子

declarator :

[pointer] direct_declarator

direct_declarator :

[memory_function_qualifier_list] identifier

(declarator)

direct_declarator [constant_expression]

direct_declarator (parameter_type_list)

direct_declarator ([identifier_list])

pointer :

[memory_function_qualifier_list] [type_qualifier_list]*

[memory_function_qualifier_list] [type_qualifier_list] pointer*

type_qualifier_list :

type_qualifier

type_qualifier_list type_qualifier

memory_function_qualifier_list :

function_qualifier

memory_function_qualifier_list memory_model_qualifier

memory_model_qualifier

memory_function_qualifier_list function_qualifier

C 言語では、値の配列、値へのポインタおよび指定する型の値を返す関数を宣言することが可能です。これらの項目は、必ず宣言子を使用して宣言します。

宣言子は、配列、ポインタまたは関数の型を宣言するために、角括弧([])、アスタリスク(*)、括弧(())などによって性質の変わる識別子です。宣言子は、ポインタ、配列、関数の宣言で見られます。

変更できない識別子で構成される宣言子は、基本型の項目の宣言に使用します。アスタリスクが識別子の左側に現れる場合は、その型は、ポインタ型に変更されます。識別子の後ろに角括弧([])が続く場合は、その型は、配列型に変更されます。識別子の後ろに括弧(())が続く場合は、その型は関数型に変更されます。

例 4.5

```
i. int table [100] ;
ii.char * cp ;
iii.long function (void) ;
```

上記の例では、(i)は、100 個の値を持つ table という名前の整数配列を宣言しています。(ii)は、文字へのポインタを宣言しています。(iii)は、引数を持たない long 型を返す関数を宣言しています。

「複合」宣言子は、複数の配列、ポインタまたは関数修飾子によって変更される識別子です。配列、ポインタ、関数修飾子のさまざまな組み合わせを 1 つの識別子に適用できます。ただし、宣言子は、次の不正な組み合わせであってはなりません。

1. 自分の要素として関数を持てない配列
2. 配列または関数を返すことのできない関数

例 4.6

```
i. int (* ((* fnarray []) ())) () ;   /*正しい*/
ii.int ((* func []) ()) [] ;          /*誤り*/
```

上記の例では、(ii)は、関数に返す整数配列を指定しているためエラーになります。

グローバルな可視性を持つ宣言においては、宣言子は任意指定です。宣言の中で宣言子が 1 つも宣言されていない場合、CCU8 では、デフォルト設定の宣言子にあたる **int** 型として扱われます。ただし、このような宣言および空の宣言に対して**/Za** オプションが指定された場合は、エラーが出力されます。

4.4.1 メモリモデル修飾子

メモリモデル修飾子は、宣言の中で使用して、明示的に変数のアドレス型を指定するために使用されます。**__near** および**__far** は、CCU8 がサポートするキーワードで、このキーワードを使用してオブジェクトのアドレス型を指定します。メモリモデル指定子は、自身の右隣のトークンに対して効力を持ちます。メモリモデル修飾子は、オブジェクトおよびオブジェクトへのポインタだけを修飾します。

例 4.7

```
int * __far fvar ; /* 'fvar' はデフォルトセグメントになくてもよいが、デフォルトセ
                      グメントにある int 型のオブジェクトを指す必要があります。 */
```

```
int __far * fptr ; /* 'fptr'はデフォルトセグメントにあり、デフォルトセグメントでな
                      くても構わない、int 型のオブジェクトを指します。 */

__far int evar ;  /* __far は型指定子を修飾できないため、エラー。 */
```

__near キーワードおよび**__far** キーワードは、データを修飾するために使用されます。オブジェクトが **const** および**__near** で修飾されている場合、ROMWINDOW 領域にこのオブジェクト用の記憶領域が割り当てられます。同様に、非 const オブジェクトが**__near** で修飾されている場合、このオブジェクト用の記憶領域は、デフォルトのデータセグメント(#0)に割り当てられます。オブジェクトが **const** および**__far** で修飾されている場合、このオブジェクト用の記憶領域は、データメモリセグメント内のテーブルセグメントのいずれかに割り当てられます。同様に、非 const オブジェクトが**__far** で修飾されている場合、このオブジェクト用の記憶領域は、データメモリセグメント内のデータセグメントのいずれかに割り当てられます。

_near および__far 指定子は、関数では使用できません。

これらのメモリモデル修飾子はすべて相互に排他的です。

構造体および共用体の場合、タグを**__near** や**__far** で修飾することはできません。**__near** や**__far** で修飾できるのは、**構造体**および**共用体**の変数だけです。構造体および共用体のタグが**__near** や**__far** で修飾されている場合、CCU8 はエラーを出力します。

例 4.8

```
struct __far tag1 {    /* エラー */
                    int a ;
                    char b ;
                  } var0 ;

struct tag {
                int a ;
                char b ;
              } var0 ;

struct tag1 var1 ;
struct tag2 __far var2 ;  /* var2 は far 構造体 */
```

上記の例では、**構造体**のタグが**__far** で修飾されているため、CCU8 は**構造体**「tag1」の宣言に対してエラーを出力します。変数「var2」は**__far** で修飾されているため、この変数はデータメモリセグメントのいずれかに割り当てられます。

構造体の各メンバーを、**__near** や**__far** で修飾することはできません。

例 4.9

```
struct tag1 {
                int       a ;
                char __far        b ;
            } var1 ;

struct tag1 var2 ;
```

上記の例では、構造体メンバー「b」の宣言に**__far** が使用されていますが、ワーニングが出力されて、この**__far** は無視されます。

typedef 識別子は、**__near** および**__far** 修飾子によって修飾することができます。

例 4.10

```
typedef int __far FVAR ;
FVAR far_identifier ;
```

上記の例では、typedef で定義された識別子「FVAR」は、**__far** によって修飾されています。このため、「FVAR」という typedef 名で宣言された変数「far_identifier」も、**__far** で修飾されます。

4.4.2 関数修飾子

CCU8 は、関数だけを修飾するキーワード**__noreg** をサポートしています。**__noreg** を使用して関数以外のオブジェクトを修飾すると、エラーが出力されます。関数が**__noreg** で修飾されている場合、CCU8 はレジスタではなく、スタックで引数を渡します。関数が複数の**__noreg** で修飾されている場合、エラーが出力されます。

例 4.11

```
int __noreg fn1 ( int arg1 ) ; /* arg1 の値はレジスタではなく、スタックを介して渡されます。 */
int fn2 ( int arg2 ) ;         /* arg2 の値は、レジスタで渡されます。 */
int __noreg var ;              /* var は関数でないため、エラー */
```

例 4.12

```
int gint ;
int __noreg fn3 (int arg3) /* __noreg で修飾されている関数定義 */
{                          /* 関数はスタックからの引数を受け取ります */
     gint = arg1 ;
}
```

4.4.3 宣言の解釈

C プログラミング言語のオブジェクト宣言構文のような宣言構文を持つ言語はありません。C 言語の複合宣言の意味は、すぐにわかるものではありません。複合宣言子は、複数の配列、ポインタまたは関数修飾子によって修飾される識別子です。

複合宣言子の解析において、角括弧と括弧(識別子右側の修飾子)は、アスタリスクの接頭語になります(識別子左側の修飾子) 。

角括弧と括弧の接頭語は同じで、左から右に結合します。宣言子の解析が完全に終了したら、最後のステップとして型指定子が適用されます。括弧を使用することで、デフォルト設定されている結合順序を上書きすることができ、これにより、指定した解析が強制的に行われます。

最も簡単な複合宣言子の解析方法は、次のステップによって、読み出す方法です。

1. 識別子から始めて、右側に進行し、角括弧または括弧を検索します(存在する場合)。
2. 発見した角括弧または括弧を解析して、進行方法を左側に変えて、アスタリスクを検索します。
3. 右括弧がいずれかの段階で発見された場合は、1 に戻って、括弧内にあるすべてのものに対して、規則 1 と 2 を適用します。
4. 最後に型指定子を適用する。

例 4.13

```
char *  ( * ( * cpvar)())[20] ;
^    ^   ^   ^ ^     ^  ^
7    6   4   2 1     3  5
```

上記の例では、ステップにラベル付けがされており、次のように解釈することができます。

1. 識別子 cpvar は、宣言されている
2. ポインタとして
3. 関数への
4. ポインタを返す
5. 20 個の要素をもつ配列への
6. ポインタとして
7. char 型の値への

int 値へのポインタ配列は、次のようにして宣言します。

例 4.14

```
int * variable [5] ;
```

次に、**int** 値配列へのポインタ宣言を示します。

例 4.15

```
int (* var) [5] ;
```

例 4.16

```
char *fn (int, char ) ;
```

例 4.16 は、**char** 型にポインタを返す関数の宣言で、Char 型は、**int** 型と **char** 型の 2 つ引数を取るように指定しています。

引数のない **float** 型を 1 つ返す関数のポインタ宣言は、次のようになります。

例 4.17

```
float (*fn1)(void) ;
```

4.5 変数宣言

ここでは、変数宣言の形式と意味を説明します。

構文:

[sc-specifier] type-specifier declarator[,declarator]

宣言方法を扱う変数の種類は、次のとおりです。

単純な変数	：単独の値を取る整数浮動少数点型の変数
構造体	：型の異なる値の集合で構成する変数
共用体	：同じ記憶空間を使用する型の異なる複数の値で構成する変数
配列	：同じ型の要素の集合で構成する変数
ポインタ	：他の変数の位置を値ではなくアドレスで持つことによって、他の変数を指す変数

4.5.1 単純な変数の宣言

構文:

[sc-specifier] type-specifier declarator [,declarator]

単純な変数の宣言では、変数の名前と型を示します。変数の記憶クラスを指定することもできます。宣言の中の識別子は、変数の名前です。型指定子は、定義されているデータ型の名前です。

コンマで区切られた識別子のリストを 1 つの宣言中に列挙すると、複数の変数を指定することができます。リストの各識別子は、変数の名前を指定します。この宣言で定義するすべての変数の型は同じです。

例 4.18

```
int x ;                          /* 単純な整数の変数を宣言しています。*/

unsigned long int lvar1, lvar2 ; /* unsigned long int 型の変数を 2 つ宣言し
                                    ています。*/

const int init = -1 ;            /* const によって修飾される 1 つの整数値変数を
                                    宣言しています。const によって -1 に初期化
                                    されます。*/

int __far fvar ;                 /* fvar を far データメモリに格納されている整
                                    数値の変数として宣言しています。*/

int __far fvar, nvar ;           /* fvar は far データメモリに格納されている整
                                    数値の変数で、nvar は near メモリにあるこ
                                    とを宣言しています。*/
```

4.5.2 構造体の宣言

構文:

struct [tag] {member-declaration-list} [declarator [,declarator]...];

struct tag [declarator[,declarator]...];

構造体の宣言では、構造体変数の名前を指定し、異なる型を持つことのできる変数の値の列(構造体のメンバーと呼びます)を指定します。構造体の型の変数は、その型によって定義されたそのままの列を保持します。

構造体の宣言は、キーワード「struct」で始まり、形式は 2 つあります。

* 最初の形式では、メンバー宣言リスト(member-declaration-list)が構造体のメンバーの型と名前を指定します。任意指定のタグ(tag)は、メンバー宣言リストで定義された構造体の型の名前を指定する識別子です。
* 2 つ目の形式は、どこか別の場所に定義された構造体の型を参照するために、以前に定義された構造体タグを使用します。このため、定義が明白な場合は、メンバー宣言リストは必要ありません。構造体の型を示す structure と typedef のポインタの宣言では、構造体の型が定義される前に構造体タグを使用できます。しかし、構造体の定義は、構造体のメンバー、typedef またはポインタを実際に使用する前に出現しなければなりません。

どちらの形式も、各宣言子が構造体変数を指定します。宣言子は、ポインタに渡す変数の型を、構造体の型、構造体変数の配列または構造体の型にポインタを返す関数に変更することができます。タグが指定されている場合で、宣言子が現れない場合、宣言は、構造体タグの型宣言を指定します。

構造体タグは、同じ可視性の他の構造体 / 共用体 / 列挙タグと区別できるようにしなければなりません。

メンバー宣言リストの引数は、複数の変数またはビットフィールド宣言を持ちます。

メンバー宣言リストで宣言した各変数は、構造体型のメンバーとして定義されます。メンバー宣言リスト内の変数宣言は、単純な変数の宣言と同じ形式です。ただし、この宣言には、記憶クラスの指定子または初期値式を含めることはできません。メンバーは、すべての変数の型、すなわち、基本型、配列、ポインタ、構造体、共用体を取ることができます。

メンバーは、自らが出現する構造体の型を持つように宣言することはできません。しかし、自らが出現する構造体の型がタグを持つ場合は、その構造体の型のポインタとして宣言することができます。このため、リンクさせた構造体リストの作成が容易に行えます。

ビットフィールド宣言の形式は次のとおりです。

type-specifier [identifier] : constant-expression;

定数式(constant-expression)には、ビットフィールドのビット数を指定します。型指定子(type-specifier)は、**unsigned char** 型または **unsigned int** 型です。指定された型が **signed char** または **signed int** の場合は、CCU8 はワーニングメッセージを出力し、unsigned として扱います。しかし、**/J** オプションが指定されている場合は、**unsigned char** 型として扱うため、**char** 型の指定したビットフィールドに対してワーニングメッセージを出力しません。**/Za** オプションが指定されている場合は、**char** 型ビットフィールドはサポートされません。

定数式は、**unsigned char** 型メンバーには 0～8 の値を取り、**unsigned int** 型メンバーには 0～16 の値を取る負でない整数値でなければなりません。ビットフィールド、ビットフィールドのポインタ、ビットフィールドに返す関数の配列は禁止されています。任意指定の識別子では、ビットフィールドの名前を指定します。指定されたビットフィールドは、ビット幅 0 を持つことはできません。名前の指定されていないビットフィールドは、位置合わせを行うためのダミーフィールドとして使用することができます。名前の指定されていないビット幅が 0 に指定されているビットフィールドは、メンバー宣言リスト内で自身に続くメンバーの格納先が確実に整数の境界から始まることを保証します。

メンバー宣言リスト中の各識別子は、リスト内では一意でなければなりません。しかし、普通の変数名または他のメンバー宣言リストの識別子と区別する必要はありません。

格納

構造体のメンバーは、宣言された順序で格納されます。最初のメンバーのメモリアドレスは一番小さく、最後のメンバーのメモリアドレスは、一番大きくなります。各メンバーの格納先は、その型に割り当てられているメモリ領域の境界からはじまります。したがって、メモリ内部では、名前の指定されていない空間(穴)が、構造体のメンバー間に生じます。

ビット列は、可能な限り小さくまとめられます。連続する **char** 型のビットフィールドメンバーは、累加したサイズが文字のサイズを超えないかぎり、同じバイト位置に格納されます。同様に、**int** 型の連続するビットフィールドメンバーは、累加したサイズが整数のサイズを超えない限り、同じワード位置に格納されます。連続する char 型ビットフィールドメンバーの合計のサイズが、新しいビットフィールドメンバーによって、文字サイズを超える場合は、新しい文字は、新しいビットフィールドメンバーとして割り振られます。同様に、連続する整数ビットフィールドメンバーの合計のサイズが、新しいビットフィールドメンバーによって、整数サイズを超える場合は、新しい整数は、新しいビットフィールドメンバーとして割り振られます。**char** 型のビットフィールドメンバーの後ろに、**int** 型のビットフィールドメンバーが続く場合、または、**int** 型のビットフィールドメンバーの後ろに **char** 型のビットフィールドメンバーが続く場合は、新しいメンバーの格納先は、次の偶数番地境界から始まります。

例 4.19

```
i. struct inv_type {
                        char item [40] ;
                        float cost ;
                        float retail ;
                        int item_on_hand ;
                        int lead_time ;
                    } inv_vara, inv_varb, inv_varc ;
```

この例は、inv_type という構造体を宣言し、変数 inv_vara、inv_varb、inv_varc を宣言しています。

```
ii. struct inv_type invntry[100] ;
```

上記の例は、inv_type 型の構造体を 100 個持つ配列を宣言しています。

```
iii.struct symbol_table {
                            char *name ;
                            int type ;
                            unsigned int scope : 2 ;
                            unsigned int sign : 1 ;
                            unsigned int qualy : 1 ;
                            struct symbol_table * next ;
                        } *global ;
```

上記の例は、symbol_table 型の構造体に渡すポインタを宣言しています。構造体は、自分自身へのポインタを持っています。また、3 つのビットフィールドと他の 2 つのメンバーを持っています。

```
iv.struct {
            unsigned char char_bit1 : 2 ;
            unsigned char char_bit2 : 4 ;
            unsigned int  int_bit1 : 8 ;
            unsigned int  int_bit2 : 1 ;
          }bit_field_str ;
```

上記の例は、char 型ビットフィールドと int 型ビットフィールドの両方を持つ構造体を宣言しています。

4.5.3 共用体の宣言

構文:

union [tag] {member-declaration-list} [declarator [,declarator]...];

union tag [declarator[,declarator]...];

共用体の宣言では、共用体変数の名前を指定し、共用体のメンバーと呼ばれる変数の値を指定します。この変数は、異なる型の値を持つことのできます。共用体型の変数は、その型の定義した値のうちの 1 つを格納します。

共用体の宣言は、構造体の宣言と同じ形式です。ただし、開始キーワードは、struct ではなく、union です。構造体と共用体の宣言には、同じ規則が適用されます。

格納

共用体の変数と関連付けられている格納先は、共用体の中で最も大きなメンバーに必要な記憶領域です。小さなメンバーが格納された場合、共用体の変数は、未使用のメモリ空間を持ちます。すべてのメンバーが同じメモリ空間に格納されます。開始アドレスはすべて同じです。格納されている値は、別のメンバーに値が割り当てられるたびに上書きされます。

共用体のすべてのメンバーは、割り当てられている記憶領域の下位の番地に配置されます。

例 4.20

```
union union_type {
                    int intvar ;
                    char charvar ;
                } union_var ;
```

上記の例は、union_type の共用体を定義し、intvar と charvar という 2 つのメンバーを持つ変数 union_var を宣言しています。

構造体または共用体のネスティングは、最大 16 層に制限されています。

4.5.4 列挙の宣言

構文:

enum [tag] {enum-list} [declarator [, declarator]...] ;

enum tag [declarator [,declarator]...] ;

列挙の宣言では、列挙変数の名前を指定し、指定された整数定数の組(列挙セット)を定義します。列挙型の変数は、その型によって定義された列挙セット値のうちの 1 つを格納します。列挙セットの整数定数は int 型です。

enum 型の変数は、int 型のように扱われます。添字式で使用されるほか、算術演算子および比較演算子のオペランドとして使用されます。

列挙の宣言は、キーワード **enum** で開始します。形式は、この冒頭に記述した 2 通りがあります。詳細は、次のとおりです。

* 最初の形式では、列挙リスト(enum-list)で列挙セット値と名前を指定します(列挙リストについては、この後、詳しく説明します)。任意指定のタグは、列挙リストで定義された列挙型の名前を指定する識別子です。宣言子は、列挙変数の名前を指定します。1 つの列挙の宣言には、複数の列挙変数を指定できます。
* 列挙の宣言の 2 つ目の形式では、どこか別の場所に定義された列挙型を参照するために、以前に定義した列挙タグを使用します。このタグは、必ず定義済の列挙型を参照しなければならず、その列挙型は、現在、可視でなければなりません。列挙型はどこか別の場所で定義されているため、列挙リストはこの種の宣言には出現しません。

列挙リストの形式は、次のとおりです。

identifier [= constant-expression]

[, identifier [= constant-expression] ...]

列挙リストの各識別子は、列挙セット値を指定します。デフォルト設定では、最初の識別子には値「0」が関連付けられており、次の識別子には値「1」が関連付けられていて、以降は同様に、最後の識別子まで順々に続きます。列挙定数の名前は、その列挙定数の値に相当します。

=付きの定数式は、値のデフォルト列を上書きします。したがって、識別子= の定数式が列挙リストに現れる場合、その識別子は、定数式が与える値と関連付けられます。定数式は、int 型でなければなりません。負になることも可能です。リストでの次の識別子は、特に明示的に指定されていない限り、定数式+1 という値と関連付けられます。

列挙セットのメンバーには、次の規則が適用されます。

* 列挙セット中の 2 つ以上の識別子を、同じ値に関連付けることが可能です。

* 列挙リスト中の識別子は、同じ可視性の他の識別子と区別がつくようにしなければなりません。普通の変数の名前と他の列挙リスト中の識別子も、区別する対象です。
* 列挙タグは、同じ可視性の他の列挙タグや構造体タグ/共用体タグとは区別できるようにしなければなりません。

例 4.21

```
enum levels_tag
{
   start,        /* value = 0 */
   primary,      /* value = 1 */
   secondary,    /* value = 2 */
   final         /* value = 3 */
} levels ;
```

この例は、levels_tag という名前の列挙型を定義し、levels という名前の変数を、その列挙型を含めて宣言しています。識別子に関連付けられている値を、次のコメントに示します。

例 4.22

```
enum constants
{
   very_low,    /* value = 0  */
   low = 10,    /* value = 10 */
   medium,      /* value = 11 */
   high = 20,   /* value = 20 */
   very_high    /* value = 21 */
} ;
const enum constants speed = high ;
```

この例では、constants という名前の列挙セット値は、speed という名前の変数に割り当てられています。列挙型 constants は、すでに宣言してあるので、この後に必要なものは、列挙タグだけです。

4.5.5 配列の宣言

構文:

type-specifier declarator [constant-expression] ;

type-specifier declarator [] ;

配列の宣言では、配列の名前を宣言し、配列の要素の型を指定します。配列の要素数も定義します。配列型の変数は、配列要素の型へのポインタとみなされます。

配列の宣言には、「構文:」で示した 2 つの形式があります。

* 最初の形式では、角括弧でくくられている定数式(constant-expression)の引数によって配列の要素数を指定します。各要素の型は、型指定子(type-specifier)から与えられる、void 型を除くすべての型になり得ます。
* 2 つ目の形式は、角括弧内の定数式が省略されています。この形式は、配列が初期化されているか、仮パラメータとして宣言されているか、プログラムのどこかで明示的に定義されている配列の参照として宣言されている場合のみ使用します。

どちらの形式でも、宣言子は変数名を指定します。また、変数の型を変更することができます。宣言子に続く角括弧([])は、配列型の宣言子を変更します。

多次元の配列は、次に示すように、括弧で囲まれた定数式のリストを持つ宣言子に基づいて宣言することができます。

type-specifier declarator[constant-expression] [constant-expression]

括弧内の各定数式は、与えられた次元の要素数を定義します。多次元配列の場合、初期化されているか、仮パラメータとして宣言されているか、プログラムのどこかで明示的に定義されている配列の参照の場合には、最初の定数式は省略できます。定数式の値がゼロの場合は、コンパイラからエラーメッセージが出力されます。

さまざまな型のオブジェクトへのポインタの配列は、複合宣言子を使用して宣言します。

格納

ある配列型に関連付けられている記憶領域は、その配列型のすべての要素に確保されている記憶領域です。1 つの配列の要素は、連続して格納され、格納位置は、最初の要素から最後の要素に向けて大きくなります。記憶領域では、配列要素は空白によって分離されません。配列は、行優先順に格納されます。たとえば、次の配列は、それぞれ列が 3 つの行 2 つで構成されています。

int list [2][3] ;

最初の行の 3 列は、2 行目の 3 列より先に格納されます。このため、最後の添字が最も早く変更されます。

制約

* 配列のサイズは、最大 65,535 バイトです。

例 4.23

```
int values [25] ;              /* それぞれが int 型の 25 個の要素を持つ values
                                  という名前の配列を宣言しています。*/
long two_dim_array [2][10] ;   /* 20 個の要素を持つ long 型の 2 次元配列を宣言し
                                  ています。*/

struct tag
{
          int ivar ;
          long lvar ;
} array_of_structures [10] ;
                           /* 10 個の要素を持つ構造体の配列を宣言しています。*/
char *arr[25] ;            /* 25 個の char ポインタの配列を宣言しています。*/
char *arr[0] ;             /* コンパイラはエラーメッセージを出力します。*/
```

4.5.6 ポインタの宣言

構文:

*type-specifier [memory_function_qualifier_list] * [modification-spec] declarator ;*

ポインタの宣言は、ポインタ変数の名前を指定し、変数が指すオブジェクトの型を指定します。ポインタとして宣言された変数は、メモリ内の番地を持ちます。

型指定子(type-specifier)は、オブジェクトの型を与えます。基本型/構造体/共用体のいずれにもなり得ます。ポインタ変数は、関数、配列、他のポインタを指すこともできます。

型指定子を **void** 型にすると、ポインタが参照する型の指定を遅らせることができます。このような項目を、**void** 型のポインタと呼びます(void *)。**void** 型のポインタとして宣言された変数は、あらゆる型のオブジェクトを指すために使用できます。しかし、ポインタの操作またはポインタが指すオブジェクトの操作を実行するためには、各操作について、ポインタの指す型が明示的に指定されていなければなりません。このような変換は、型変換によって施されます。

変更指定子(modification-spec)は、**const** または **volatile** のどちらか、または両方です。これらは、それぞれ、このポインタはプログラムで変更しないように(**const**)指定するか、プログラムの制御を超える何らかのプロセスによって、適正な手段で変更されるように(**volatile**)指定します。

例 4.24

```
char * volatile * const buffer ;
/* 'buffer'は、常に同じ内容を持つ ROM 内の 1 つの場所です。しかし、'buffer'によって
示された場所の内容は、プログラムの制御範囲を超えて変更が行われます。*/
```

const 変更指定子も、ROM に入るポインタを修飾します。ポインタ宣言中の間接表現レベルは、その間接表現が ROM 内の場所を指す場合は、const として修飾されます。

例 4.25

```
int * const ptr ;          /* ptr は、RAM の位置情報を持つ ROM 内の 1 つの場所です。
                              */
int const * const iptr ;   /* iptr は、ROM 内の位置情報を持つ ROM 内の 1 つの場所
                              です。*/
```

宣言子は、変数の名前を指定します。型変更子を持つことができます。たとえば、宣言子がある配列を表現する場合、ポインタの型は、配列へのポインタの型に変更されます。

格納

あるアドレスに必要な記憶領域は、選択するメモリモデル(memory_function_qualifier_list)によって異なります。ポインタが near メモリを指している場合は、ポインタのサイズは 2 バイトです。ポインタが far メモリを指している場合は、ポインタのサイズは 3 バイトです。

例 4.26

```
char __near * string ;           /* string は、near 文字を指すポインタです。
                                    */
long __far *arr_of_pntrs [10] ;  /* far long 型のポインタの配列です。*/
```

4.6 関数の宣言とプロトタイプ

構文:

[sc-specifier] [type-specifier] declarator([declarator] [[,declarator]...])

関数プロトタイプともいわれる関数の宣言では、関数の戻り型と名前を設定し、型、仮パラメータの名前および関数に渡す引数の数を指定することができます。関数の宣言は、関数本体を定義しません。単純に関数をコンピュータに知らせる情報を作成します。この情報は、関数の呼び出しを確実に行うために、コンパイラが引数の実際の型を検査することを可能にします。

関数呼び出しの中の括弧で括られた引数に先行する式が、識別子だけで構成されていて、この識別子に可視の宣言がない場合、識別子は、関数呼び出しを持つブロックの最も内側のブロックにあるかのように宣言されます。この間接的な宣言は、そのブロックのみ可視です。

記憶クラス指定子(sc-specifier)は、**extern** または **static** のどちらでも指定できます。

型指定子(type-specifier)は、関数の戻り型を与えます。宣言子は、関数の名前を指定します。型指定子が省略されている場合は、その関数の返す値は **int** 型とみなされます。

仮パラメータについては、第 4.6.1 章の説明を参照してください。最終的な宣言リスト(declaration-list)は、同じ行についてさらに詳しい宣言を表現します。現在の関数と同じ型の値を返す別の関数を宣言したり、最初の関数の戻り型と同じ型の変数を宣言したりすることができます。このような宣言は、それぞれ、コンマで区切ります。

4.6.1 仮パラメータ

仮パラメータは、関数に渡す実パラメータのことです。関数の宣言においては、パラメータの宣言は、実引数の数と型を指定します。仮パラメータの識別子を含めることができます。これらのパラメータの宣言は、コンパイラが関数定義を処理する前に出現する関数呼び出しで検査される引数に影響します。

一部の仮パラメータのリストは、上記の構文で宣言できます。仮パラメータリストには、最低でも 1 つは宣言子を含めます。「省略形式」によって、1 つのコンマとその後に 3 つのドット(,...)を記述して示すため、さまざまな数のパラメータを指定できます。関数は、最後のコンマの前にある宣言子または識別子の型の数だけの引数を持つことが期待されています。

例 4.27

```
int function1 (int number_of_items,...) ;
```

構造体または共用体の変数も、関数に渡す実引数にすることができます。仮パラメータリストは、構造体または共用体のパラメータも含めることができます。

例 4.28

```
int function2 (struct a_tag arg, union v_tag value) ;
```

プロトタイプ宣言で仮パラメータを指定するために使用する識別子は、記述のみで、宣言の後、有効範囲から外れます。この理由から、この種の識別子は、関数定義の宣言部で使用する識別子と同じでなくてかまいません。同じ名前を使用すると、読みやすくなりますが、それ以上の意味はありません。

4.6.2 戻り型

関数は、関数と配列を除くあらゆる型の値を戻すことができます。

例 4.29

```
struct tag
{
       int a ;
       long b ;
} input_structures[10] ;

struct tag
get_structure (int value)
{
    return (input_structures [value]) ;
}
```

4.6.3 仮パラメータのリスト

関数宣言に続く括弧内に表記する仮パラメータリスト(formal-parameter-list)引数のあらゆる要素は、任意です。

構文:

[type-specifier] [declarator[[,...][,...]]]

関数定義で、仮パラメータが省略されている場合は、括弧には **void** というキーワードを記述して、その関数に渡す引数は存在しないことを指定します。括弧を空白のままにした場合は、この関数に引数が渡されるのかどうか検査が行われ、引数の型に関するチェックは行われません。

仮パラメータリストの中の宣言には、記憶クラスの **auto** を示す指定子だけを含めることができます。型指定子が含まれている場合は、基本型またはポインタ型について型名を指定することができます。宣言子は、型指定子と識別子に適した変更子を組み合わせて構成します。他に、抽象宣言子という、特定の識別子を持たない宣言子を使用することができます。抽象宣言子については、第 4.10 章を参照してください。

例 4.30

```
void function (void) ;          /* 戻り値と引数を持たない関数を宣言しています。
                                   */

long fn (int, char) ;           /* int 型と char 型の引数を 2 つ渡し、long 型の
                                   戻り値を返す関数を宣言しています。*/

char *strtok(char s[],char c) ; /* 文字のポインタを返し、文字配列と文字の 2 つの
                                   引数を渡す関数を宣言しています。*/

struct inv_type *sfn () ;       /* 構造体 inv_type のポインタを返す関数を宣言
                                   しています。引数の数と型は未定義です。*/
```

4.6.4 関数修飾子

CCU8 は、関数だけを修飾するキーワード**__noreg** をサポートしています。関数が**__noreg** で修飾されると、CCU8 は、レジスタによってではなく、スタックによって引数を渡します。関数識別子の**_noreg** が関数定義の中で指定されている場合、CCU8 は、レジスタからではなくスタックから引数を受け取ります

4.7 記憶クラス指定子

変数の記憶クラスは、項目のグローバルまたはローカルを決定します。グローバルな項目は、プログラムの実行が終了するまで存続します。

すべての関数は、グローバルです。

ローカルの変数は、自分の定義されているブロックに実行の制御が移るたびに新しい記憶領域を割り当てられます。実行の制御がブロックになくなると、変数は意味のない値になります。

CCU8 には、次の 5 つの記憶クラス指定子が用意されています。

1. auto
2. static
3. extern
4. typedef
5. register

auto 記憶クラス指定子と共に宣言される項目はローカルです。**static** または **extern** 指定子と共に宣言される項目はグローバルです。

typedef 指定子は、記憶領域を確保しませんが、構文の都合上から記憶クラス指定子と呼ばれています。詳細は、第 4.9.2 章を参照してください。

register 記憶クラス指定子が指定されている場合、コンパイラは、可能な場合には、変数をレジスタに格納します。レジスタへの格納は、アクセス時間を短縮し、コードのサイズを小さくします。**register** 記憶クラスで宣言された変数は、**auto** 変数と同じです。

コンパイラが register 宣言を発見したときに、レジスタが使用できない場合は、その変数には、**auto** 記憶クラスが適用され、**auto** 記憶クラスの規則に基づいて処理されます。**register** として宣言された変数については、アドレス演算子(単項&) を適用できません。

例 4.31

```
register int count ;
register index ;
```

記憶クラス指定子には、明確な意義があります。なぜなら、記憶クラス指定子は、関数と変数および、その記憶クラスの可視性に影響するからです。可視性という用語は、関数または変数を名前で参照することのできるソースプログラムの部分を意味します。グローバルを持つ項目は、ソースプログラムの実行が終了するまで存続しますが、プログラムのあらゆる場所で可視というわけではありません。

変数宣言および関数宣言のソースファイル内への配置も、記憶クラスと可視性に影響します。すべての関数定義に含まれない宣言は、外部レベルで現れます。関数定義の中にある宣言は、内部レベルで現れます。

個々の記憶クラス指定子の正確な意味は、次の 2 つの要素に依存します。

* 宣言は外部レベルで現れるか内部レベルで現れるか
* 宣言する項目は変数か関数か

次では、各種宣言における記憶クラス指定子の意味を説明します。変数宣言または関数宣言で記憶クラス指定子 が省略された場合のデフォルト動作についても説明します。

4.7.1 外部レベルでの変数宣言

外部レベル(すべての関数の外)での変数宣言では、**static** および **extern** 記憶クラス指定子を使用するか、記憶クラス指定子自体を省略します。**auto** 記憶クラス指定子は、外部レベルでは使用できません。

外部レベルでの変数宣言は、変数の定義(宣言の定義)または別の場所で定義されている変数への参照(宣言の参照)のどちらかです。

変数を初期化する外部変数の宣言は、変数の宣言で定義します。

外部レベルでの定義の形式は、次のように何通りかあります。

* **static** 記憶クラス指定子と共に宣言される変数は、その変数の定義です。**const** および非 const の **static** 変数は、定数式で初期化できます。たとえば、**static int** x;と **const static int** y = 10; は、変数「x」と「y」の定義とみなします。
* 外部レベルで明示的に初期化される変数は、その変数の定義です。CCU8 は、外部レベルで **const** および非 const の変数の初期化を許可しています。たとえば、**const int** i = 10、**int** y = 20 は、それぞれに変数「i」と「y」の定義です。

変数が外部レベルで定義されると、自分が出現するソースファイルの定義以降の部分で可視になります。同じソースファイルの中でも、自分自身が定義される前は、可視ではありません。次の方法で、参照宣言を使用して可視にしなければ、同じプログラムの他のソースファイルでは、不可視です。

外部レベルでの変数定義は一度だけしか行えません。静的記憶クラスが指定されていて、**static** クラスが別のソースファイルにある場合は、他の変数を同じ名前で定義することができます。各 **static** 定義は、自分の記述されているソースファイルの中でだけ可視なので、競合しません。

extern 記憶クラス指定子は、別の場所で定義された変数への参照を宣言します。**extern** 宣言は、定義を他のソースファイルの中で可視にしたり、同じソースファイルの中で、変数自身が定義される前にその変数を可視にしたりするために使用します。**extern** 宣言は、宣言の発生するソースファイルの残りの部分では常に可視です。

extern 参照が効力を持つには、参照する変数を外部レベルで一度だけ定義します。定義は、プログラムを構成する任意のソースファイルの中に含めることができます。

記憶クラス指定子と初期値式の両方を外部レベルの変数定義で省略するのは、特殊なケースです。たとえば、宣言 **int** a;は、有効な外部宣言です。この宣言は、文脈によって、2 つの異なる意味を持ちます。

* プログラムの別の場所に同じ名前の変数の外部宣言がある場合、現在の宣言は、**extern** 記憶クラス指定子が宣言の中で使用されたかのように定義している変数への参照とみなされます。
* 変数の外部宣言が、プログラムの中にない場合は、宣言した変数には、リンク時に記憶領域が割り当てられます。この種の変数を共有変数と呼びます。1 つのプログラムの異なるソースファイルにこの種の宣言が複数現れる場合には、記憶領域は、その変数に宣言された最大サイズが割り当てられます。たとえば、file1 は宣言 **int** i;を持ち、file2 は宣言 **long** i;を持ち、file1 と file2 が同じプログラムの構成要素の場合、リンク時に、**long** 値用の記憶領域が「i」に割り当てられます。

例 4.32

```
/* FILE1 */
extern int global_variable ;  /*下に定義される global_variable への参照です。
                                  */
main ()
{
     global_variable = global_variable + 100 ;
     file1_function () ;
}
int global_variable ;         /* global_variable の定義です。*/


file1_function ()
{
     file2_function () ;
     global_variable -= 100 ;
}
/* FILE2 */
extern int global_variable ;  /* global_variable への参照です。*/
static int file2variable ;    /* file2variable の定義です。 file2variable
                                  は FILE2 でだけ可視です。*/
file2_function ()
{
      global_variable += 10 ;
      return ;
}
```

4.7.2 内部レベルでの変数定義

内部レベルの変数宣言では、記憶クラス指定子の **auto**、**extern** および **static** を使用できます。記憶クラス指定子がこの宣言で省略された場合のデフォルトの記憶クラス指定子は **auto** です。

ローカルの記憶クラス指定子は、ローカルな変数を宣言します。**auto** 変数は、自分が宣言されているブロックの中でだけ可視です。**auto** 変数の定義には、初期値式を含めることができます。**auto** 変数は、自動的には初期化されないため、明示的に初期化を行うか、ブロックの中で文を使用して初期値を割り当てます。初期化していない auto 変数の値は、定義されません。

static ローカル変数は、あらゆる外部項目または **static** 項目で初期化できますが、非 static の **auto** 項目では初期化できません。なぜなら、auto 項目のアドレスは一定ではないからです。

内部レベルで静的記憶クラスと共に宣言された変数はグローバルですが、可視性は、自分が宣言されたブロックの中でだけ有効です。**auto** 変数とは異なり、**static** 変数は、ブロックから抜けても値を保持し続けます。**const** で修飾された static 変数は、プログラムの実行を開始するときに一度だけ初期化されます。ブロックに制御が移るたびに初期化されることはありません。

extern 記憶クラス指定子と共に宣言される変数は、プログラムを構成するソースファイルのいずれかの中で外部レベルとして同じ名前で定義されている変数への参照です。内部 **extern** 宣言は、外部レベルの変数の定義を、同ブロックの中で可視にするために使用します。外部レベルで宣言されていない場合は、内部レベルでキーワード「**extern**」によって宣言された変数は、自分が宣言されているブロックの中でだけ可視です。

例 4.33

```
/********* FILE1 **********/
main ()
{
      extern int a ;    /* FILE2 で定義されている‘a’への参照です。*/
        static int b ;  /* グローバルで関数の中でだけ可視です。*/
        int c = 0 ;     /* デフォルトの記憶クラスは auto で、制御がこの関数に移る
                           たびにゼロに初期化されます。*/
      file2 () ;
}
/************FILE2 ************/
int a ;
int c ;
file2 ()
{
      int a ;                /* グローバルの‘a’の再定義で、現在は不可視です。*/
      static int * d = &c ;  /* グローバル ‘c’のアドレスは、‘d’ を初期化するため
                                に使用されます。初期化は、関数に制御が移っても、
                                毎回行われはしません。実行開始直後にだけ行われ
                                ます。*/
```

```
        a = c ;
}
```

4.7.3 内部レベルおよび外部レベルでの関数宣言

関数宣言は、**static** または **extern** 記憶クラス指定子のどちらも持つことができます。関数は常にグローバルです。

関数に対する可視性の規則は、変数とは、次のような点で異なります。

* **static** で宣言される関数は、その関数を定義しているソースファイルの中でだけ可視です。同じソースファイル中の関数は、**static** 関数を呼び出すことができますが、他のソースファイル中の関数を呼び出すことはできません。別のソースファイル中にある同じ名前の別の **static** 関数とは、競合しないため使用できます。
* **extern** として宣言される関数は、後に static として再宣言されていない限り、プログラムを構成するすべてのソースファイルに対して可視です。すべての関数は、**extern** 関数を呼び出すことができます。
* 記憶クラス指定子を省略した関数宣言は、デフォルトで **extern** として扱われます。

4.8 初期化

構文:

= initializer

変数は、変数宣言の中で宣言子に初期値式(initializer)を適用することによって、初期値を設定することが可能です。初期値式の値(複数対応)が、変数に割り当てられます。初期値式の前に等符号(=)を記述します。

初期化の規則は、次のとおりです。

* 外部レベルで宣言された const および非 const で修飾された変数は、初期化することができます。**const** で修飾された変数が外部レベルで初期化されていない場合、この変数には、値「0」が割り当てられます。
* auto 記憶クラス指定子で宣言された変数は、自分が宣言されているブロックに 制御が移るたびに初期化されます。auto 変数の宣言で初期値式の記述が省略されている場合は、この変数の初期値は未定義です。集合(配列、構造体および共用体)および非集合の変数を初期化できます。

* 外部変数宣言およびすべての static 変数の初期値には、外部変数であるか内部変数であるかにかかわらず、定数式を使用します。auto 変数の初期化には、定数値および変数値のいずれも使用できます。
* const 修飾子は、項目の ROM への配置も行います。

例 4.34

```
char *volatile input_buf ;
const int integer_var1, integer_var3 ;
int integer_var2 ;
long long_var = 4 ;                              /*正しい*/
const char * error_ptr = “pointer” ;             /*正しい*/
const char * const ptr = “pointer”;              /*正しい*/
char * volatile * const buffer = &input_buf ; /*正しい*/
const int * var_ptr = &integer_var1 ;            /*正しい*/
int * const var_ptr1 = &integer_var2 ;           /*正しい*/
const int * const var_ptr2 = &integer_var3 ;  /*正しい*/
```

次では、基本変数、ポインタ、集合型変数を初期化する方法を説明します。

4.8.1 基本変数とポインタ型変数

構文:

= expression

式(expression)の値を、変数に代入します。代入するための変換規則を適用します。詳細は、第 5.31 章を参照してください。

内部で宣言された **static** 変数は、定数値による初期化のみ可能です。外部宣言された変数または **static** 変数のアドレスは、一定なので、内部的に宣言された static ポインタ変数の初期化に使用することができます。しかし、**auto** 変数のアドレスは、初期値式として使用することはできません。なぜなら、ブロックを実行するたびに値が変わるからです。ただし、**/Za** オプションを指定すると、extern 変数を初期化できません。

例 4.35

```
long lv = 100 ;                          /* lv は、定数値 100 に初期化されます。*/
static const int * const scp = 0 ;       /* ポインタ scp はゼロに初期化されます。
                                            */
int x ;
int * const y = &x ;                     /* ポインタ y は、x のアドレスで初期化さ
                                            れます。*/
int z = 10;                              /* データメモリ変数 z は、10 に初期化され
                                            ます。*/
const int m ;                            /* デフォルトでは､m は 0 に初期化されます。
                                            */
func ()
{
    int local1 = 10 ;                    /* 適正な初期化です。*/
    static int local = 100 ;
    int *p = &z ;                        /* 有効です｡グローバル変数のアドレスを初
                                            期化で使用できます。*/
    static int *const lp = &local1 ;/* 無効です。ローカル変数のアドレスは
                                            static 変数の初期化には使用できませ
                                            ん。*/
}
```

4.8.2 集合型

構文:

= {initializer-list}

初期値式リスト(initializer-list)は、各初期値式がコンマで区切られたリストです。各初期値式は、定数式または初期値式リストです。このため、中括弧に囲まれた初期値式リストは、他の初期値式リストの中に現れることができます。この形式は、集合型のまとまったメンバーを初期化するときには便利です。

各初期値式リストでは、定数式の値が、対応する集合変数のメンバーに、順番に割り当てられます。共用体を初期化する場合、共用体の最初のメンバーには、初期値式の値が割り当てられます。

初期値式リストに、集合型の数より少ない数の値しかない場合は、残りのメンバーまたは集合型の要素には領域が確保されます。初期値式リストに、集合型の数より多い数の値がある場合は、エラーになります。これらの規則は、各初期値式リストに適用されるほか、集合体全体に適用されます。

例 4.36

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

上記の例では、サイズの指定がなく、3 つの初期値式の記述があるので、「x」を 3 つのメンバーを持つ 1 次元の配列として宣言し、初期化します。

例 4.37

```
long y [4][3] = {
                    {1, 4, 7},
                    {2, 5, 8},
                    {3, 6, 9},
                } ;
```

これは、中括弧をひとまとめにして初期化を行います。1、4 および 7 は、配列 y[0]の最初の行、すなわち、y[0][0]、y[0][1]、y[0][2]を初期化します。同様に、次の 2 行は、y[1]と y[2]を初期化します。初期値式が先に終了したので、y[3]には領域が確保されます。次の記述でも、これとまったく同じ効果が得られます。

例 4.38

```
long y [4][3] = { 1,4,7,2,5,8,3,6,9 } ;
```

「y」の初期値式は、左中括弧で始まりますが、このリスト全体が y[0]についての初期化ではありません。y[0]の初期化には、リストの 3 つの要素が使用されます。同様に次の 3 つが連続的に、y[1]と y[2]に適用されます。

初期化

例 4.39

```
const long y [4][3] = { {1}, {2}, {3}, {4} } ;
```

配列の最初の列、すなわち y[0][0]、y[1][0]、y[2][0]、y[3][0]を、それぞれ、1、2、3、4 でを初期化します。残ったものには領域を確保します。変数は **const** で修飾されているので、残りは、ゼロに初期化されます。

4.8.3 文字列の初期値式

構文:

= *“characters”*

文字配列は、次のように、文字列リテラル(“characters”)で初期化できます。

例 4.40

```
char str_arr [] = “abc” ;
```

この例は、str_arr を 4 つの要素を持つ文字配列として初期化します。4 つ目の要素は、すべての文字列リテラルを終了させるヌル文字です。配列のサイズが指定されていて、文字列が指定された配列サイズより長い場合は、余分な文字は無視されて、ワーニングメッセージが表示されます。たとえば、次の宣言は、str_arr を 3 つの要素を持つ文字配列として初期化します。

例 4.41

```
const char str_arr[3] = “abcd” ;
```

文字列の最初の 3 文字だけが「str_arr」に代入されます。文字の「d」とヌル文字によって終了する文字列も切り捨てられます。これにより、終りのない文字列が作成され、その旨、警告メッセージが出力されます。文字列が指定されている配列サイズより短い場合は、配列の残りの要素には、領域が確保されます。

4.9 型の宣言

構造体または共用体の宣言は、構造体または共用体の名前とメンバーを定義します。宣言された型の名前は、その型を参照する変数宣言または関数宣言の中で使用することができます。これは、宣言する多数の変数や関数が同じ型の場合に便利です。

typedef 宣言は、1 つの型の型指定子を定義します。typedef 宣言は、プログラマが宣言した型によって、または型へ、すでに定義されている型の名前を短くしたり、もっと意味のある名前にするときに使用したりすることができます。

4.9.1 構造体型と共用体型

構造体型と共用体型の宣言は、これらの型の変数宣言と同じ共通の形式です。しかし、構造体と共用体の型宣言は、次の方法が違います。

* 構造体と共用体の型宣言では、変数は省略します。
* 構造体と共用体の型宣言タグは必須です。タグは、構造体と共用体の型の名前を指定します。
* メンバーの型を宣言するメンバー宣言リストは、構造体と共用体の型宣言に必ず出現しなければなりません。

例 4.42

```
struct tag {
                int x ;
                char arr[20] ;
            } ;
```

上記の例は、tag という名前を持つ構造体型を宣言しています。

4.9.2 Typedef 宣言

構文:

typedef type-specifier declarator[,declarator]...;

typedef 宣言は、記憶クラス指定子に代わってキーワードの「**typedef**」を使用することを除けば、変数宣言に似ています。**typedef** 宣言は、宣言で指定された型を仮定するのではなく、変数または関数の宣言と同じ方法で解釈され、その型の同義語になります。

typedef 宣言は型を作成しません。既存の型の同義語を作成するか、他の方法で指定されている型の名前を作成します。ポインタ型、関数型、配列型を含むすべての型は、**typedef** で宣言できます。**typedef** 名は、構造体または共用体の型を指すポインタとしても宣言できます。

例 4.43

```
typedef int fixed_point ; /* 'fixed_point' は、'int'の同義語です。したがって、
                             fixed_point x ;の宣言は、int x ;を宣言しているのと同じです。*/
typedef struct {
                    float y ;
                    long x ;
                } COMPLEX ;
COMPLEX *sp ;
```

2 つのメンバーを持つ COMPLEX を構造体型として宣言しています。COMPLEX は、この先の宣言の中で使用できます。

COMPLEX *sp ; /*COMPLEX 構造体のポインタ sp を宣言しています。*/

4.10 型名

型名は、普通の変数宣言と型宣言に加えて、特定のデータ型を指定します。型名は、次の 3 種類の状況で使用されます。

* 関数宣言(プロトタイプ)の仮パラメータリストの中
* 型キャスト
* sizeof 演算子

仮パラメータリストに関する詳細は、第 4.6.1 章を参照してください。

基本型、構造体、共用体の型名は、単純にこれらの型の型指定子です。ポインタ型、配列型、関数型の型名の形式は次のとおりです。

type-specifier abstract-declarator

抽象宣言子(abstract-declarator)は、識別子を持たない宣言子で、ポインタ、配列、関数の 1 つまたは複数の変更子で構成されます。ポインタ変更子(*)は、常に宣言の中の識別子に先行します。配列([])と関数(())の変更子は、常に、識別子に続きます。これを踏まえると、識別子をどこに置けばよいのかを決定でき、さらに宣言子を解釈できるようになります。

抽象宣言子は、複雑です。複雑な抽象宣言子中の括弧は、特定な解釈をします。ちょうど、宣言の中の複合宣言子に対して行うのと同じです。typedef 宣言で指定する型指定子は、型名とみなされます。

例 4.44

```
int * ;          /* int のポインタの型名です。*/
long (*)[5] ;    /* long 型要素の配列のポインタの型名です。*/
int (*)(void) ;  /* 引数がなく int 型を返す関数のポインタの型名です。*/
```

4.11 関数

関数は、宣言と文で構成される独立した集合体で、通常、特定のタスクを実行するように設計されます。C プログラムは、必ず 1 つは関数を持ちます。この関数を「**main**」関数と呼びます。この他に関数を持つことができます。ここからは、C 関数の定義、宣言、呼び出しを行う方法を説明します。

4.11.1 関数の定義

構文:

[sc-specifier][type-specifier] declarator([formal-parameter-list])

function-body

[sc-specifier][type-specifier] declarator([identifier-list])

[parameter-declarations]

function-body

[sc-specifier] [type-specifier] declarator([declarator] [,declarator]...])

function-body

関数の定義は、関数の名前、仮パラメータ、本体を指定します。さらに、関数の戻り型と記憶クラスも明記します。

4.11.1.1 記憶クラス

関数定義の記憶クラス指定子(sc-specifier)は、関数に **extern**(外部)記憶クラスまたは **static**(静的)記憶クラスを与えます。関数定義に記憶クラス指定子が含まれていない場合は、記憶クラス指定子のデフォルトの **extern** 記憶クラスが指定されます。

静的記憶クラスを持つ関数は、自分の宣言されているソースファイルの中でだけ可視です。他の関数は、**extern** 記憶クラスが明示的または暗示的に与えられているかどうかにかかわらず、プログラムを構成するソースファイル全体で可視です。

静的記憶クラスを使用したい場合は、関数の最初の宣言が発生するとき(存在する場合)および関数を定義するときに宣言しなければなりません。

4.11.1.2 戻り型と関数名

関数の戻り型は、関数が返す値のサイズと型を指定します。関数定義の構文中の型指定子に相当します。あらゆる基本型を指定できます。型指定子が含まれていない場合、戻り型は **int** 型になります。

宣言子は、関数識別子で、ポインタ型に変更できます。宣言子に続く括弧は、項目を関数として指定します。

関数定義で与えられる戻り型は、プログラムの別の場所にあるその関数の宣言で指定されている戻り型と一致しなければなりません。関数の戻り型は、関数が値を返す場合にだけ使用できます。関数は、式を含む return 文が実行されたときに値を返します。式は、評価された後、必要な場合は、関数の戻り値の型に変換され、関数が呼び出されたポイントへ戻ります。実行する return 文がない場合、または、return 文に式が含まれていない場合は、戻り値は定義されません。

キーワードの「**void**」が型指定子として使用されている場合は、関数は値を返すことができません。

4.11.1.3 仮パラメータ

構文:

形式 1:

[sc-specifier][type-specifier] declarator([formal-parameter-list])

function-body

形式 2:

[sc-specifier][type-specifier] declarator([identifier-list])

[parameter-declarations]

function-body

仮パラメータは、関数呼び出しで関数に渡された値を受け取る変数です。構文の形式 1 では、関数名に続く括弧には、仮パラメータの完全な宣言が含まれています。仮パラメータリスト(formal-parameter-list)は、コンマで区切られた仮パラメータ宣言の列です。

関数定義の形式 2 では、仮パラメータは、関数の本体(function-body)の直前にある右括弧に続けて宣言されています。この形式の任意指定である識別子リスト(identifier-list)は、関数が仮パラメータの名前として使用する識別子のリストです。リストの中での識別子の順序は、関数呼び出しの中で識別子が値を取る順序付けになります。識別子リストは、ゼロ個または複数の識別子で構成されます。各識別子の間は、コンマで区切ります。この識別子リストは、中が空であっても、括弧の中に指定しなければなりません。パラメータ宣言(parameter-declaration)は、形式 2 の中では、識別子の型を指定します。

引数が何も渡されない場合は、仮パラメータのリストはキーワードの「void」に置き換えることができます。

仮パラメータ宣言は、仮パラメータに格納される値の型、サイズ、識別子を指定します。形式 2 では、これらは、他の変数宣言と同じ形式になっています。形式 1 では、仮パラメータリスト中の各識別子は、自分に適した型識別子を先行させる必要があります。

例 :4.45

```
/* 関数は形式 1 で定義されています。*/
void form1 (long a, long b, long c)
{
     return ;
}
/* 関数は形式 2 で定義されています。*/
void form2 (a,b,c)
```

```
long b,c ;
long a ;
{
     return ;
}
```

仮パラメータが存在している場合は、仮パラメータの順序と型は、すべての関数宣言の中で同じでなければなりません。関数を呼び出す場合の実引数の型は、対応する仮パラメータの型と互換のある指定を行うようにします。仮パラメータは、基本型またはポインタ型を持つことができます。

仮パラメータに使用が認められている記憶クラスは **auto** だけです。関数名に続く括弧のなかに、宣言されていない識別子のデフォルトの型は **int** 型です。

仮パラメータの識別子は、関数に渡す値を参照する関数の本体の中で使用します。これらの識別子は、関数の本体の中、最上部では再定義できません。しかし、内側のブロックの中では再定義することができます。

形式 2 では、識別子リストに出現する識別子だけを、仮パラメータとして宣言できます。また、仮パラメータを宣言する順序には、特別な規則はありません。

コンパイラは、必要な場合には、各パラメータについて、通常の算術変換を実施します。変換後は、**char** 型の仮パラメータはなくなります。なぜなら、**char** 型で宣言されたすべての仮パラメータは、**int** 型に変換されているためです。

4.11.1.4 関数の本体

関数の本体は、この関数が何をするのか定義する文から構成されます。これらの文が使用する変数の宣言も含まれている可能性があります。

関数の本体で宣言されたすべての変数は、特に指定がなければ、auto 記憶クラスを持ちます。関数が呼び出されると、ローカル変数用の記憶領域が作成されます。式を含む return 文は、関数が値を返す場合には、関数の本体の中で実行しなければなりません。

4.11.2 関数のプロトタイプ

関数のプロトタイプ宣言は、関数の名前、戻り型、記憶クラスを指定します。すべての引数または一部の引数の型と識別子も指定します。プロトタイプを定義する形式は、右側の括弧の直後にセミコロンを付けて終了するという点を除けば、関数定義と同じです。このため、本体はありません。

関数呼び出しが宣言または定義より先に出現する場合は、コンパイラがデフォルトの関数のプロトタイプを作成して、「**int**」型の戻り型を与えます。使用する引数の型と数は、仮パラメータを宣言する際の基準です。したがって、関数の呼び出しは、宣言としての意が暗に含まれていますが、作成されるプロトタイプは、これから先の関数の呼び出しや定義を適切に代理することはできません。この間接的な宣言は、この関数呼び出しを含んでいるブロックのみ有効です。

プロトタイプは、関数の属性を指定して、自分の定義が先行する関数の呼び出しの引数と戻り型の不適当な組み合わせについてチェックできるようにします。プロトタイプで静的記憶クラスが指定されている場合、その静的記憶クラスは関数定義でも指定されていなければなりません。

関数のプロトタイプは、次の重要な役割を果たします。

* int 型以外の型を返す関数の戻り型を指定します。このような関数が定義または宣言の前に呼び出された場合は、結果は定義されません。
* プロトタイプがパラメータの型に関する完全なリストを持っている場合は、関数の呼び出しまたは関数定義の中で出現する引数の型を検査できます。プロトタイプの宣言の中にあるパラメータリストは、関数呼び出し時に使用する実引数と、関数定義の中にある仮パラメータとの対応状況をチェックするために使用されます。
* プロトタイプは、関数が定義される前に関数に渡すポインタを初期化するために使用されます。

4.11.3 関数の呼び出し

構文:

expression([expression-list])

関数の呼び出しは、制御と実引数(存在する場合)を関数に渡す式です。関数の呼び出しでは、式で関数のアドレスを算出します。式リスト(expression-list)は、式をコンマで区切って記載しているリストです。これらの式の値は、関数に渡される実引数です。関数に引数がない場合の式リストは空です。

関数を実行すると:

1. 式リストの中にある式が評価され、通常の算術変換によって変換されます。関数のプロトタイプが利用できる場合、式の結果は、仮パラメータの宣言と整合させるために変換されます。
2. 式リストの中にある式は、呼び出された関数の仮パラメータに渡されます。リストの最初の式は、必ずその関数の最初の仮パラメータに対応し、2 番目の式は、2 番目の仮パラメータに対応し、リストの終了まで、順番に進んで行きます。呼び出された関数は、実引数のコピーを使用するため、引数を変更しても、コピー元の変数の値は、影響を受けません。
3. 実行の制御が関数の最初の文に移ります。

4. 関数の本体で、return 文が実行されると、呼び出し元の関数に制御が戻り、値を返します。return 文が実行されない場合は、呼び出された関数の完全な実行後、呼び出し元の関数に制御が戻ります。このような場合は、戻り値は定義されません。

4.11.3.1 実引数

実引数は、基本型またはポインタ型を使う値であれば、どんな値でもなり得ます。すべての実引数は、値で渡されます。ポインタは、関数が値にアクセスする手段として参照する方法を提供します。

関数呼び出しの式は、評価後、次のように変換します。

* 関数呼び出しの実引数について、通常の算術変換を実行します。プロトタイプが利用できる場合は、導き出された引数の型とプロトタイプの対応する仮パラメータを比較します。一致しない場合は、両方の変換を行い、診断メッセージを出力します。
* プロトタイプが利用できない場合は、値を関数に渡す前に、各実引数について、デフォルトの変換を実行します。デフォルト変換では、「**char**」型の引数は「**int**」型に変換し、「**float**」型の引数は「**double**」型に変換します。仮パラメータの型が変換後の実引数の型と一致するパラメータを持つプロトタイプを作成します。

式リスト中の式の数は、関数のプロトタイプまたは関数定義中の仮パラメータの数と一致していなければなりません。プロトタイプの仮パラメータリストにキーワードの「**void**」しかない場合は、コンパイラは、現在の関数呼び出しおよび関数定義の引数はゼロであるものと解釈して、診断メッセージを表示します。

4.11.3.2 再帰的呼び出し

C プログラムの関数は、すべて再帰的に呼び出すことができます。つまり、自分自身を呼び出すことができます。C コンパイラは、自分自身に対する再帰的呼び出しを無限に実行することを許容します。関数が呼び出されると、毎回、新しい記憶領域を仮パラメータと auto 変数に割り当てて、以前の値や終了していない呼び出しを、上書きしないようにします。**static** として宣言された変数には、各再帰的呼び出しのたびに新しい格納先は確保されません。これらの格納先は、プログラムの生存期間中は常時存在しています。

4.12 ASM 宣言

キーワードの**__asm** は、次の形式の「ASM」文を指定する場合に使用します。

__asm (文字列)

上記の文は、関数の内側および外側で記述することができます。関数の内側および外側での「**__asm**」文は似ています。詳細は、第 6.8 章を参照してください。

5. 式と演算子

5.1 演算子

演算子は、値を生成するために使用する何らかの意味を持つ連続するシンボルです。最も単純な式は、定数と変数名です。通常、式は演算子と部分式を組み合わせて、値を生成します。

C 言語の演算子は、単純な変数の識別子と定数を組み合わせて複合式を構成します。C 言語の演算子は、次のように分類することができます。

* 単項演算子 : オペランドを 1 つ取ります。
* 2 項演算子 :2 つのオペランドを取り、さまざまな算術演算および論理演算を行います。
* 条件付き演算子(3 項演算子) : 3 つのオペランドを取り、最初の式の評価結果に基づいて、2 つ目または 3 つ目の式のどちらかを評価します。
* 代入演算子 : 変数に値を代入します。代入する前に右辺の値を左辺の値の型に変換します。
* コンマ演算子 : コンマで区切られた式の評価を左から右に行い、結果は最右の式に示します。

単項演算子は、自身のオペランドの前に出現し、右から左に結合します。2 項演算子は、左から右に結合します。C 言語には 3 項演算子が 1 つあり、右から左に結合します。

C 言語の演算子の優先順位と結合規則は、式中のオペランドのグループ化と評価に影響します。演算子の優先順位は、優先順位の高いまたは低い他の演算子が存在する場合には重要な意味を持ちます。高い優先順位を持つ演算子を持つ式が先に評価されます。

下表は、C 演算子の優先順位と結合規則を示しています。優先順位の高い演算子から低い演算子へ上から順番に並んでいます。同じ行に並んでいる複数の演算子は、同等の優先順位を持つため、評価する順序はそれぞれの結合規則に基づいて決定します。

C 演算子の優先権と結合性 ：

演算子	結合規則
() [] -> .	左から右へ
- + ~ ! * & ++ -- sizeof casts	右から左へ
* / %	左から右へ
+ -	左から右へ
<< >>	左から右へ
< <= > >=	左から右へ
== !=	左から右へ
&	左から右へ
^	左から右へ
|	左から右へ
&&	左から右へ
| |	左から右へ
?:	右から左へ
= += -= *= /= %= &= ^= |= <<= >>=	右から左へ
,	左から右へ

１つの式に、同じ優先順位を持つ複数の演算子を含むことが可能です。複数の同等の演算子が 1 つの式の中で同じレベルで出現した場合は、その演算子の結合規則に基づいて右から左へ、または、左から右へ評価されます。

5.2 左辺値と右辺値

変数識別子は、C の基本的な式の 1 つです。この種の式は、変数のオブジェクトとして 1 つの値を作成します。しかし、他の演算子とともに変数識別子を使用すると、式では、メモリ上の変数の位置が評価されます。変数のアドレスを左辺値と呼びます。このアドレスに格納されているオブジェクトを右辺値と呼びます。CCU8 は、次の式で変数の右辺値と左辺値を評価して使用します。

```
x = y;
```

変数 y の内容は、変数 x に代入されます。つまり、代入を行う際、右辺の式は、式の右辺値を生成し、左辺の式は式の左辺値を生成します。

次の C 言語の式は、左辺値の式です。

* スカラー変数の識別子
* スカラー要素の参照
* 構造体変数および共用体変数の参照
* 構造体メンバーおよび共用体メンバーの参照(左辺値以外のフィールドへの参照を除く)
* ポインタの参照（参照先ポインタとも呼びます。アスタリスク（*）に続けてアドレスの値を持つ式)
* 括弧で囲まれた上記の各式

上記の式は、次の左辺値の構文で表現します。

左辺値：

identifier

expression[expression]

expression.expression

expression->expression

**expression*

(左辺値)

すべての左辺値の式は、メモリ内の 1 ヵ所を表現します。

5.3 変換

演算子の中には、オペランドによっては、ある型から別の型へ値の変換を行うものがあります。ここでは、このような変換から期待される結果について説明します。

5.3.1 整数への格上げ

整数を使用する式では、必ず次のうちの 1 つを使用します。

1. 文字
2. 整数または文字のビットフィールド
3. 列挙型のオブジェクト

1 つの int 型で元の型の値をすべて表現できる場合は、値は **int** 型に変換されます。int 型では表現できない場合は、**unsigned int** 型に変換されます。これを、整数への格上げと呼びます。整数への格上げを行っても、他のすべての算術型は変更されません。

5.3.2 算術変換

多くの演算子は、変換を発生させ、同様の方法で結果を生成します。結果的に、オペランドを共通の型にします。この型は、生じる結果と同じ型です。このパターンを通常、算術変換と呼びます。

* はじめに、一方のオペランドが **long double** 型の場合、他方は **long double** 型に変換されます。
* これ以外の場合で、一方のオペランドが **double** 型の場合、他方は **double** 型に変換されます。
* これ以外の場合で、一方のオペランドが **float** 型の場合、他方は **float** 型に変換されます。
* これ以外の場合で、両方のオペランドに整数への格上げを実行後、一方のオペランドが **unsigned long int** 型の場合、他方は **unsigned long int** 型に変換されます。
* これ以外の場合で、一方のオペランドが **long int** 型で、他方が **unsigned int** 型の場合、両者は **unsigned long int** 型に変換されます。
* これ以外の場合で、一方のオペランドが **long int** 型の場合、他方は **long int** 型に変換されます。
* それ以外の場合で、一方のオペランドが **unsigned int** 型の場合、他方は **unsigned int** 型に変換されます。
* それ以外の場合は、両者のオペランドは **int** 型です。

5.3.3 ポインタの変換

2 つのポインタの操作中、両方のサイズが同じになるように変換されます。ポインタのサイズは、メモリモデルとそのポインタに指定されているメモリモデル修飾子によって決まります。ポインタが**__far** で修飾されている場合、ポインタのサイズは 3 バイトです。ポインタが**__near** で修飾されている場合、ポインタのサイズは 2 バイトです。

式は、near ポインタ(2 バイト)と__far ポインタ(3 バイト)の両方を含むことができます。サイズの異なる 2 つのポインタの操作中、2 つのポインタは同じサイズになるように拡張されます。near ポインタは、上位バイトにデフォルトセグメントアドレスを持つ far ポインタに変換されます。

5.4 一次式と演算子

単純な式を一次式と呼びます。一次式(primary_expression)には、識別子、定数、文字列または括弧で囲まれた式があります。

primary_expression :

identifier

constant

string

(expression)

5.4.1 識別子

識別子(identifier)は、変数または関数の名前を指定します。変数は、プログラムで操作する基本データオブジェクトの 1 つです。宣言には、使用する変数を列挙し、変数の型も指定します。

識別子は、識別子の左隣にキーワード**__far** を指定すると、far 変数として修飾できます。far 変数は、デフォルトセグメント内に格納する必要はありません。このため、セグメントの切り替えは far 変数にアクセスする前に行われます。

5.4.2 定数

定数(constant)オペランドは、自分が表現する定数の型と値を持ちます。型は、形式に依存します。文字定数は **int** 型です。列挙型定数も **int** 型です。一般に、定数の型は、**int** 型、**unsigned int** 型、**long** 型、**unsigned long** 型、**float** または **double** 型です。

5.4.3 文字列

文字列リテラル(string)は、文字または二重引用符で囲まれた隣接する文字の列です。空白文字によってのみ区切られた 3 つ以上の隣り合う文字列リテラルは、1 つの文字列リテラルとして結合されます。結合後、ヌルバイト「**\0**」を末尾に追加して、プログラムに文字列の終了位置を示します。

文字列リテラルは、コードメモリに char 型の要素の配列として格納されます。文字列リテラルの型は、最初は「const char 配列」です。これが、通常は「const char のポインタ」に変更され、結果として文字列中の最初の文字へのポインタを生成します。文字列リテラルは、静的記憶クラスを使用します。

5.4.4 括弧で囲まれた式

括弧で囲まれた式(expression)は、括弧のない式の型および値と同じ式を持つ一次式です。主に、括弧は、演算子の結合規則と優先順位を変更するために使用します。

例 5.1

```
(5 + 5) * 3
```

上記の例では、5 + 5 を囲む括弧は、乗算演算子(*)の左側のオペランドである値 5 + 5 を意味しています。上記の式の結果は 30 です。括弧がない場合、式は 5 + 5 * 3 となり、結果は 20 です。

5.5. 配列参照

array_reference :

expression1 [expression2]

括弧演算子([および])は、配列の要素を表すために使用されています。角括弧で囲んだ別の式が続く式は、添字のある配列参照であることを示しています。

2 つの式の一方は、「T へのポインタ」型でなければなりません。T は何らかの型を意味します。他方は、整数型でなければなりません。添字式の結果の型は T です。

式：expression1[expression2] は、*((expression1) + (expression2))と書き換えた場合も、定義上の意味は同じです。これは、どちらの場合も、expression1 から expression2 が配置されている先のアドレスの値を表しているためです。

5.6 関数呼び出し

function calls :

expression ( )

expression (argument_list)

関数は、括弧の続く式です。括弧には、コンマで区切られた引数が入るか空です。引数リスト(argument_list)の構文は、次のとおりです。

argument_list :

expression

argument_list, expression

関数呼び出しは、宣言が先行する場合と宣言のない場合とがあります。宣言が先行していない場合は、戻り型は、「int」型であると想定されます。

関数呼び出しの中の式は、何らかの型 T のための、「T を返す関数のポインタ」型でなければなりません。関数呼び出しの結果の型は T です。

関数呼び出しの準備として、各引数のコピーが作成され、すべての引数の渡し方は、必ず値で行われます。呼び出される関数は、自分のパラメータの値を変更します。しかし、この変更は、呼び出し側の関数の引数値には影響しません。

引数は、渡す前に整数への格上げを受けます。

関数呼び出しの引数の数が、関数の定義の中で指定している引数の数と一致しない場合、パラメータリストの最後に省略記号 (...)の記述がなければ、エラーメッセージが表示されます。省略記号の記述がない場合、引数の数はパラメータ数と同じか、パラメータ数より多くなければなりません。明示的に入力されているパラメータより多い引数の存在は、デフォルト引数の助長を引き起こします。

関数にプロトタイプの指定がなく、本体が定義されていない場合、上記したチェックは実行されません。CCU8 は、プロトタイプの指定がなく、本体定義が指定されている場合は、この定義からプロトタイプの代理をします。プロトタイプと本体の定義が異なる場合は、プロトタイプは本体定義で上書きされます。

引数の評価順序は、左から右です。あらゆる関数の再帰的呼び出しは、認められています。

実際には near ポインタを受け取る関数に far ポインタが引数として渡されると、引数ポインタのセグメント情報は失われます。near ポインタの位置に far ポインタが引数として渡されると、CCU8 がエラーを出力します。逆に、実際には far ポインタを受け取る関数に near ポインタが渡された場合は、ワーニングメッセージが出力されます。near ポインタは、ポインタの上位バイトにデフォルトセグメントアドレスを持つ far ポインタに変換されます。

次の組み込み関数がサポートされています。

__EI　　__DI

上記の組み込み関数のプロトタイプは、次のとおりです。

```
void __EI (void) ;

void __DI (void) ;
```

これらの組み込み関数は、他の関数と同様に呼び出すことができます。

5.7 構造体参照と共用体参照

structure_reference :

expression . identifier

expression -> identifier

構造体または共用体のメンバーは、ドット(.) または右矢印(→)のどちらかの演算子で参照できます。

ドット演算子

後ろにドットおよび識別子(identifier)が続く式(expression)は、構造体または共用体のメンバーを参照します。最初のオペランド式は、構造体または共用体でなければなりません。識別子はその構造体または共用体のメンバー名を指定しなければなりません。

結果の値は、構造体または共用体の指定されたメンバーです。結果の型は、メンバーと同じ型です。結果式は、メンバーの型が配列型でなければ、左辺値です。

矢印演算子

後ろに矢印および識別子が続く式も、構造体または共用体のメンバーを参照します。最初のオペランド式は、構造体または共用体のポインタでなければなりません。識別子は、そのポインタの指す構造体または共用体のメンバー名を指定しなければなりません。

結果は、そのポインタの指す構造体または共用体の指定したメンバー名への参照です。結果の型は、目的のメンバーの型と同じです。

例 5.2

```
struct example {
                    int member1 ;
                    int member2 ;
                    struct example * ptr_to_struct ;
                } s_variable, struct_array [10] ;
1. s_variable.ptr_to_struct = &s_variable ;
2. (s_variable.ptr_to_struct)->member1 = 25 ;
3. struct_array [7].member2 = 100 ;
```

上記の例では、

1. s_variable 構造体のアドレスが構造体の ptr_to_struct メンバーに代入されます。
2. ポインタ式 s_variable.ptr_to_struct は、ポインタ選択演算子(->) と一緒に使用して、メンバー(member1)に値を代入します。
3. 単一の構造体メンバーは、構造体の配列から選択されます。

5.8 後置インクリメント

post_increment :

expression ++

後置インクリメントは、式の後ろに演算子「++」が続く場合に実行されます。結果の値は、オペランドの値です。値を使用した後、オペランドに 1 が追加されます。結果の型はオペランドの型です。式の結果は、左辺値がなくなったものです。

オペランドは、整数型、浮動小数点型、ポインタ型のいずれかで、変更が可能な (非定数)左辺値の式でなければなりません。整数型または浮動小数点型のオペランドは、整数値 1 で増値されます。ポインタ型のオペランドは、自身が指すオブジェクトのサイズ分のアドレスで増値されます。増分されたポインタは、次のオブジェクトを指します。

例 5.3

```
int a, b ;
a = b ++;
```

上記の例では、「b」の値が「a」に代入された後、「b」がインクリメントされます。

5.9 後置デクリメント

post_decrement :

expression --

後置デクリメントは、式の後ろに演算子「--」が続く場合に実行されます。結果の値は、オペランドの値です。値を使用した後、オペランドから 1 が引かれます。結果の型はオペランドの型です。式の結果は、左辺値がなくなったものです。

オペランドは、整数型、浮動小数点型、ポインタ型のいずれかで、変更が可能な (非定数)左辺値の式でなければなりません。整数型または浮動小数点型のオペランドは、整数値 1 で減値されます。ポインタ型のオペランドは、自身が指すオブジェクトのサイズ分のアドレスで減値されます。減分されたポインタは、前のオブジェクトを指します。

例 5.4

```
int a, b ;
a = b -- ;
```

上記の例では、「b」の値が「a」に代入された後、「b」がデクリメントされます。

5.10 前置インクリメント

pre_increment:

++ expression

式の前に演算子「++ 」がある場合は、単項式です。オペランドは、1 で増値(++)されます。式の値は、インクリメントした後のオペランドの値です。オペランドは、左辺値でなければなりません。他の規則は、後置インクリメントと同様です(詳細は、第 5.8 章を参照してください)。

例 5.5

```
int a, b ;
a = ++ b ;
```

「b」の値を代入する前にインクリメントされます。インクリメントした値が「a」に代入されます。

5.11 前置デクリメント

pre_decrement:

-- expression

式の前に演算子「--」がある場合は、単項式です。オペランドは、1 でデクリメントされます。式の値は、デクレメントした後のオペランドの値です。オペランドは左辺値でなければなりません。他の規則は、後置デクリメントと同様です(詳細は、第 5.9 章を参照してください)。

例 5.6

```
int a, b ;
```

```
a = -- b ;
```

「b」の値を代入する前にデクリメントされます。デクリメントした値が「a」に代入されます。

5.12 アドレス演算子

address_operator :

& expression

アドレスは、アドレス演算子「&」で計算します。単項演算子の「&」は、自分のオペランドのアドレスを取ります。オペランドは、代入演算の有効な左辺値を使用します。しかし、ビットフィールドおよび register として宣言されているオブジェクトは、禁止されています。

配列名がアドレス演算子のオペランドの場合、ワーニングメッセージが表示されます。配列名はアドレスなので、&演算子が無視されます。関数指名子がアドレス演算子のオペランドの場合には、ワーニングは出力されません。&演算子は、関数指名子には無視されます。

結果は、左辺値オペランドへのポインタです。オペランドの型が T の場合、結果は、「T へのポインタ」です。

例 5.7

```
int x, * p ;
p = &x ;
```

アドレス演算子(&) は、x のアドレスを取り、p に代入します。

アドレスが、far セグメントに常駐する変数用として取得された場合、アドレスのサイズは 3 バイトです。アドレスは、下位 2 バイトにオフセット値、上位バイトにセグメントアドレスを持ちます。

例 5.8

```
int __far x ;
int __far * p ;
p = &x ; /* x のアドレスのサイズは 3 バイトです。*/
```

5.13 間接演算子

indirection_operator :

** expression*

単項の*演算子は、間接表現を示すもので、ポインタの再参照に使用されます。オペランドはポインタの値です。

演算の結果は、オペランドが示す値、つまりオペランドが指定するアドレスの値です。結果の型はオペランドのアドレスの型です。式の型が「T へのポインタ」の場合は、結果の型は T です。

オペランドが配列型でなければ、結果は左辺値です。

例 5.9

```
int x, * p ;
x = * p ;
```

間接演算子(*)は、p に格納されているアドレスにある整数値にアクセスするときに使用します。アクセスした値を整数 x に代入します。

5.14 単項プラス演算子

unary_plus_operator :

+ expression

「+」演算子のオペランドは、算術型でなければなりません。結果は、オペランドの値です。整数への格上げが実行されます。結果の型は、生成されたオペランドの型です。

5.15 単項マイナス演算子

unary_minus_operator :

- expression

単項マイナス(-)演算子は、負のオペランドを生成します(2 の補数)。「-」演算子のオペランドは、算術型でなければなりません。結果は負のオペランドです。整数への格上げが実行されます。負のゼロはゼロです。結果の型は、生成されたオペランドの型です。

例 5.10

```
int variable ;
variable = 999 ;
variable = - variable ;
```

変数の値は負の 999 です(「-999」のことです)。

5.16　1 の補数演算子

bit_not_operator :

~ expression

演算子 ~ は、演算子オペランドのビット単位の補数を生成します。演算子 ~ のオペランドは、整数型でなければなりません。結果は、この演算子のオペランドの 1 の補数です。整数への格上げが実行されます。結果の型は、生成されたオペランドの型です。

例 5.11

```
unsigned int a, x ;
a = 0xaaaa ;
x = ~a ;
```

X に代入される値は、符号なしの値「0xaaaa」の 1 の補数です(「0x5555」のことです)。

5.17 論理否定演算子

logical_not_operator :

! expression

論理比較は、演算子 ！ が先行する式で行われます。オペランドは、算術型またはポインタ型でなければなりません。

結果の値は 1 または 0 です。オペランド比較結果がゼロのとき、結果は 1 になり、オペランドがゼロではないときに結果は 0 になります。結果の型は **int** 型です。

例 5.12

```
int x, y ;
if (! ( x < y) )
   fn () ;
```

x が y と等しいか y より大きい場合、式の結果は 1(true)です。x が y より小さい場合、結果は 0(false)です。

5.18 SIZEOF 演算子

sizeof_operator :

sizeof expression

sizeof (typename)

sizeof 演算子は、自身のオペランドの型のオブジェクトを格納するために必要なバイト数を生成します。オペランドは、評価の行われない式または括弧で囲まれた型名です。

sizeof 演算子が char 型に適用された場合の結果は 1、配列に適用された場合の結果は、配列の総バイト数です。要素数が「n」の配列のサイズは、要素 1 つ分のサイズの「n」倍です。

sizeof 演算子が構造体または共用体に適用された場合、結果は、構造体または共用体をメモリ境界に配置するために使用するパディングを含む、オブジェクトのバイト数です。

sizeof 演算子は、不完全な型のオペランドへの適用は認められていません。

結果は、符号なし整数の定数です。結果の型は、符号なし整数です。

型名(typename)は、構文の都合上、名前を省略するオブジェクトの型宣言です。

例 5.13

```
long array [10] ;
z = sizeof ( array ) ;
```

z の値は 40 です。

Sizeof 演算子は、式にも適用できます。適用される式は評価されません。結果は式の結果のサイズです。

例 5.14

```
int a, x, y, z ;
x = 10 ;
y = 10 ;
z = sizeof ( x = (y *2) ) ;
a = x ;
```

上記の例では、「x = (y * 2)」は評価されません。したがって、値「10」は「a」に代入されます。「20」ではありません。

例 5.15

```
int i, j ;
int * dptr ;
fn ()
{
       i =  sizeof (dptr) ;
}
```

上記の例では、ポインタ変数「dptr」のサイズは、コンパイルするときのデータメモリモデルによって決まります。プログラムを near メモリモデルでコンパイルした場合は、「i」の値は 2 です。far メモリモデルでコンパイルした場合は、「i」の値は 3 です。

5.19 キャスト演算子

cast_operator :

(typename) expression

キャスト演算子は、括弧で囲まれたデータ型名で構成されます。括弧で囲まれた型名の先行する単項式の値は、指定されている型に変換されます。型名(typename)に関する詳細は、第 4.10 章を参照してください。

オペランドが変数の場合、データの型は指定されている型に変換されます。変数の内容は変更されません。

型名のサイズが式のサイズより大きい場合は、キャスト式の結果は、左辺値ではありません。

near 変数は、far 変数にキャストできません。しかし、near メモリのポインタは far メモリのポインタにキャストできます。far メモリのポインタが near メモリのポインタに代入された場合は、CCU8 のワーニングが出力されます。near メモリのポインタを far メモリのポインタにキャストした場合は、ポインタの変換が実行されます。

例 5.16

```
int x, y ;
int * cvar ;
int __far * dvar ;

dvar = ( int __far *) cvar ; /* cvar は、far ポインタとしてキャストされ、dvar
                                に代入されます。*/
```

5.20 乗法演算子

multiplicative_expression :

*expression * expression*

expression / expression

expression % expression

乗法演算子は、*、/、%です。これらの演算子は、左から右に適用します。

*および/のオペランドは、算術型でなければなりません。%のオペランドは、整数型でなければなりません。オペランドに対して通常の算術変換が行われ、結果の型を予測します。

2 項式の*演算子は、乗算を意味します。

最初のオペランドを 2 番目のオペランドで除算するとき、2 項式の/ 演算子は、商が求められ、% 演算子は、余りが求められます。2 番目のオペランドがゼロの場合は、結果は定義されません。CCU8 が 2 番目のオペランドがゼロであることを検出すると、ワーニングメッセージを表示します。

例 5.17

```
unsigned int x, y, i, n, j ;
y = x * i ;
n = i / j ;
n = i % j ;
```

5.21 加法 演算子

additive_expression :

expression + expression

expression - expression

加法演算子 + と - は、左から右に適用します。オペランドが算術型の場合は、通常の算術変換が行われます。

+ 演算子の結果は、オペランドの和です。オブジェクトのポインタおよびあらゆる整数型の値を追加できます。CCU8 は、1 つのオブジェクトのサイズを計算し、整数で乗算し、オフセット値を求めます。次に、オフセット値を指定された要素のアドレスに加えます。結果は元のポインタと同じ型のポインタです。元のオブジェクトからの適切なオフセット値にある別のオブジェクトを指します。したがって、P があるオブジェクトへのポインタで、式 P+1 が次のオブジェクトへのポインタです。

CCU8 は、2 つのポインタの加算が行われると、エラーを出力します。

演算子 - の結果は、オペランドとオペランドの差です。あらゆる整数型の値をポインタから減算することができます。これを行う場合は、加算を行う場合の変換と条件の規則を適用します。

同じ型のオブジェクトへのポインタの 2 つで減算すると、結果として、ポインタが指していた 2 つのオブジェクト間の距離を表す符号付き整数が得られます。結果の値は、2 つのポインタの差を求め、ポインタが指すオブジェクトのサイズで差を除算した計算値です。

2 つのポインタを減算すると、ポインタへのオフセット値だけが引かれます。型の異なるオブジェクトを指しているポインタの組み合わせは認められていません。

例 5.18

```
int x, y, i, j, k, l ;
char *p1, *p2 ;
long array1 [20], array2 [20] ;

y = x + i ;
p1 = p2 + 2 ;
j = &array1[k] - &array1[l] ;
```

5.22 シフト演算子

shift_expression :

expression << expression

expression >> expression

シフト演算子<<と>>は、左から右へ適用されます。どちらの演算子のオペランドも整数でなければなりません。整数への格上げの実行対象です。

結果の型は、生成された左側のオペランドの型です。

e1 << e2 の結果は、左に e2 ビット分シフトされた式 e1 の値です。CCU8 は、空いたビットにはゼロを詰めます。

e1 >> e2 の結果は、右に e2 ビット分シフトされた式 e1 の値です。CCU8 は、空いたビットには、左側のオペランド e1 が符号なしの場合は、ゼロを詰めます。それ以外の場合は、e1 の符号ビットを複写して詰めます。

右側のオペランドが負の場合、または左側の式の型のビット長より大きいか等しい場合は、結果は定義されません。

例 5.19

```
unsigned int x, y, z ;
x = 0x00aa ;
y = 0x5500 ;
z = ( x << 8) + ( y>>8) ;
```

上記の例では、「x」は左に 8 ビットシフトされ、「y」は右に 8 ビットシフトされます。シフトされた値は、0xaa55 であり、「z」に代入されます。

5.23 関係演算子

relational_expression :

expression < *expression*

expression > *expression*

expression <= *expression*

expression >= *expression*

関係演算子は、より小さい(<)、より大きい(>)、小さいか等しい(<=) および大きいか等しい(>=) です。

算術オペランドについて、通常の算術変換が行われます。関係式が false の場合の結果は 0、関係式が true の場合の結果は 1 です。結果は int 型です。

同じ型のオブジェクトのポインタ同士は比較できます。結果は、ポインタが指しているオブジェクトのアドレス空間の相対位置によって決まります。ポインタの比較は、同じオブジェクトの部分に対してのみ定義されます。2 つのオブジェクトが同じ単純なオブジェクトを指す場合は、それらを比較したときは同じとなります。ポインタが同じ構造体のメンバーを指す場合は，構造体の後のほうで宣言されているオブジェクトへのポインタのほうが高いです。また、ポインタが同じ共用体のメンバーを指すときには、それらは等しくなります。一方、ポインタが配列の要素を参照するときには、その比較は対応する添字の比較と同じです。

ポインタは、値 0 を持つ整数定数式や void のポインタと比較できます。ただし、**/Za** オプションが指定されている場合は、このような比較は認められません。

比較演算子は、左から右へ適用します。a < b < c は (a < b) < c と解析されます。a < b は、0 または 1 のいずれかに評価されます。

例 5.20

```
const static int x = 10, y = 10 ;
```

```
int z ;
z = x > y ;
```

x と y は等しいので、値 0 は z に代入されます。

例 5.21

```
char array [10], *p ;
void * v_ptr ;
for (p = array ; p < &array[10] ; p ++ )
*p = '\0' ;
if ( v_ptr > p ) /*/Za オプションを指定した場合は、エラーが出力されます。*/
   array [2] = '\0' ;
```

上記のプログラムは、配列の各要素をヌル文字定数に初期化します。

5.24 等値 演算子

equality_expression :

expression == expression

expression != expression

== (等しい)および!= (等しくない)演算子は、優先順位が低いことを除けば、関係演算子と同じです。したがって、a<b と c<d が同じ true の値を持つ場合は、a<b == c<d は、1 です。

等値演算子には、関係演算子の規則が適用されます。両方のオペランドがポインタの場合は、ポインタ変換が行われます。

等値演算子の使用例として 2 つの有効な関数を示します。

例 5.22

```
strcmp ( char s[], char t[])
{
    int i = 0 ;
    while ( s[i] == t[i] )
         if ( s[i ++] == '\0' )
             return (0) ;
    return ( s[i] - t[i] ) ;
}
```

上記の関数は、等値演算子を使用しています。上記の関数は、「s」が「t」より小さい場合は、負の値を返します。「s」は「t」と等しい場合は、ゼロを返します。「s」が「t」より大きい場合は正の値を返します。

例 5.23

```
squeeze( char s[], int c )
{
    int i, j ;
    for (i=j=0; s[i] != '\0'; i++)
      if ( s[i] != c )
          s [j++] = s [i] ;
    s [j] = '\0' ;
}
```

上記のプログラムは、文字列「s」から文字「c」をすべて削除します。

5.25 ビット論理積演算子

bit_and_expression :

expression & expression

ビット論理積演算子 (&)は、整数のオペランドのみ扱います。通常の算術変換が行われます。

結果は、オペランドの対応するビット論理積関数です。ビット論理積演算子は、最初のオペランドの各ビットと 2 つ目のオペランドの対応ビットを比較します。両方のビットが 1 の場合は、結果ビットは 1 です。それ以外の場合は、結果ビットは 0 です。

5.26 ビット排他的論理和演算子

bit_xor_expression :

expression ^ expression

ビット排他的論理和演算子(^)は、整数のオペランドのみ扱います。通常の算術変換が行われます。

結果は、オペランドの対応するビット排他的論理和関数です。ビット排他的論理和演算子は、最初のオペランドの各ビットと 2 つ目のオペランドの対応ビットを比較します。一方のビットがゼロで他方のビットが 1 の場合、結果ビットは 1 です。それ以外の場合は、結果ビットは 0 です。

5.27 ビット論理和演算子

bit_or_expression :

expression | *expression*

ビット論理和演算子（|）は、整数のオペランドのみ扱います。通常の算術変換が行われます。

結果は、オペランドの対応するビット論理和関数です。ビット論理和演算子は、最初のオペランドの各ビットと 2 つ目のオペランドの対応ビットを比較します。一方のビットが 1 の場合、結果ビットは 1 です。それ以外の場合は、結果ビットは 0 です。

5.28 論理積演算子

logical_AND_expression :

expression && expression

演算子 &&は、論理積演算に使用します。演算子&&は、左から右へ適用されます。式の結果は、1 または 0 です。結果は int 型です。

比較するオペランドは、同じ型である必要はありませんが、算術型またはポインタ型でなければなりません。

CCU8 が左側のオペランドだけを調査して評価できる場合は、右側のオペランドは評価しません。

式 e1 && e2 では、最初のオペランド e1 が評価されます。あらゆる副作用も検討されます。この式がゼロと等しい場合、式の値はゼロで、2 つ目の式 e2 は評価されません。式がゼロではない場合は、e2 が評価されます。ゼロと等しい場合は、結果はゼロです。それ以外の場合は 1 です。

5.29 論理和演算子

logical_OR_expression :

expression || *expression*

演算子 || は、論理和演算に使用します。演算子|| は左から右へ適用されます。式の結果は 1 または 0 です。結果は int 型です。

比較するオペランドは、同じ型である必要はありませんが、算術型またはポインタ型でなければなりません。

CCU8 が左側のオペランドだけを調査して評価できる場合は、右側のオペランドは評価しません。

式 e1 || e2 では、最初のオペランド e1 が評価されます。あらゆる副作用も検討されます。この式がゼロではない場合は、式の値は 1 で、2 つ目の式 e2 は評価されません。e1 がゼロの場合は、e2 が評価され、ゼロと等しい場合は、結果はゼロです。それ以外の場合は 1 です。

例 5.24

```
xor ( int a, int b )
{
    return ( (a || b) && ! (a && b)) ;
}
```

XOR 演算は、上記の関数で実行します。XOR 関数は、一方のオペランドだけが true の場合(非ゼロ)には true 値(1)を返します。上記の関数は、論理積演算子と論理和演算子の両方を示しています。

5.30 条件式および条件演算子

conditional_expression :

expression1 ? expression2 : expression3

C 言語には、3 項演算子が 1 つあります。条件演算子 (?:)です。

式 1(expression1)は、整数型、浮動小数点型、ポインタ型でなければなりません。0 との等価性について評価されます。評価は次のように行われます。

式 1 が 0 と等しくない場合は、式 2(expression2)を評価します。式の結果は式 2 の値です。

式 1 が 0 と等しい場合は、式 3(expression3)を評価します。式の結果は式 3 の値です。

式 2 または式 3 のどちらかだけを評価します。演算子? :は、右から左に適用されます。

条件演算の結果の型は、次のように式 2 または式 3 の型によって決まります。

* 式 2 または式 3 が整数または浮動小数点型の場合は、演算子は通常の算術変換を行います。結果の型は、変換後のオペランドの型です。
* 式 2 と式 3 の両方が同じ構造体、共用体またはポインタ型の場合、結果の型は、構造体、共用体またはポインタの型と同じです。
* 両方のオペランドが void 型の場合は、結果の型は void です。
* どちらかのオペランドが、オブジェクトへのポインタの場合(どの型のオブジェクトでも同じ)で、他方のオペランドが void へのポインタの場合は、オブジェクトへのポインタは、void のポインタに変換され、結果は void へのポインタになります。
* 式 2 と式 3 のどちらかが near ポインタで、他方が far ポインタの場合、ポインタ変換が実行されます。
* 式 2 と式 3 のどちらかがポインタで、他方のオペランドが値 0 を持つ定数式の場合、結果の型はポインタ型です。

例 5.25

```
int i, j ;
j = ( i < 0 ) ? (-i) : (i) ;
```

上記の例では、i の絶対値が j に代入されています。i が 0 より小さい場合は、j に-i が代入されます。i が 0 より大きいか、0 と等しい場合は、i は j に代入されます。

5.31 代入式および代入演算子

assignment_expression :

expression assign_op expression

代入演算子はいくつかありますが、すべて左から右に適用されます。代入演算子は、次のとおりです。

= += -= *= /= %= >>= <<= &= |= ^=

すべての演算子は、左側のオペランドとして左辺値を必要とします。左辺値は、変更可能な値でなければなりません。配列は認められません。不完全な型または関数は使用できません。また、const で修飾したオペランド型は禁止されています。代入式の型は、式自身の左側のオペランドの型です。値は、代入が行われた後の左側のオペランドの格納された値です。

＝を使った単純な代入では、式の値が、左辺値で参照されるオブジェクトの値と置き換わります。次のいずれかが true でなければなりません。

* オペランドは両方とも算術型である。この場合、右側のオペランドは代入により左側のオペランドの型に変換されます。
* オペランドの 1 つがポインタで、他方は void へのポインタである。
* 左側のオペランドはポインタで、右側のオペランドは値 0 を持つ定数式である。
* 両方のオペランドが関数または右側のオペランドに const あるいは volatile がないこともあるという点を除けば、その型が同じであるオブジェクトへのポインタです。
* far メモリのポインタが near メモリのポインタに代入された場合、セグメント情報は消失します。そのポインタを使用してさらに演算を行うと、プログラムが予定外の動作をする可能性があります。CCU8 では、far ポインタが near ポインタに代入されるとエラーメッセージが出力されます。ただし、near ポインタを far ポインタに代入する場合は、セグメント情報は失われません。デフォルトセグメントアドレスが near ポインタの上位バイトに代入されます。near ポインタが far ポインタに代入されると、CCU8 はワーニングメッセージを出力します。

e1 op= e2 の形の式は、e1 = e1 op e2 と書き換えることができます。

例 5.26

```
float y ;
int x ;
y = x ;
```

x の値は、浮動小数点型に変換され、y に代入されます。

例 5.27

```
# define MASK 0Xff00
unsigned int n ;
n &= MASK ;
```

上記の例では、「n」および「MASK」についてビット論理積演算が実行されます。結果は「n」に代入されます。

5.32 コンマ式およびコンマ演算子

comma_expression :

expression, expression

コンマ演算子（,）は、2 つのオペランドを左から右へ連続的に評価していきます。演算結果は右側のオペランドと同じ値、同じ型になります。あらゆる型がオペランドとして使用できます。コンマ演算子は、オペランド間の型変換を行いません。

コンマ演算子は、通常は、1 つの式しか認められていない箇所で、相互に関連する複数の式を使用する場合を評価します。

例 5.28

```
1. f (a, (t =3, t +2), c) ;
2. for (i = 0,j = 0; i< 10; i ++, j +=2)
        array[i] = j ;      /*偶数の配列です。*/
```

例 1 の 2 つ目の引数の値は、5 です。

コンマ演算の結果が配列の場合は、ポインタに変換されます。

5.33 定数式

定数式は、定数を評価するすべての式のことです。定数式のオペランドとして使用できるのは、整数定数、文字定数、浮動小数点定数、型変換、sizeof 式、その他の定数式です。演算子は、演算子使用して、修飾および結合することができます。

前処理指令で使用される定数式は、特定の制約を受けます。sizeof 式の使用、任意の型への型変換、浮動小数点定数の使用は認められていません。

浮動小数点定数を含む定数式、非算術型および式のアドレスへの型変換は、初期値式の中でだけ認められています。単項のアドレス演算子(&)は、外部レベルで宣言された基本型の変数または添字付きの配列参照に適用することができます。

6. 文

6.1 概要

ここでは、C 言語の文について説明します。文は、明示的に制御を別の場所に移す場合以外は、記述されている順序で実行されます。

文は、効力を有効にするために実行されます。値は持ちません。文は、次のグループに分類されます。

statements :

labeled_statement

expression_statement

compound_statement

selection_statement

iteration_statement

jump_statement

asm_statement

制約：複文、条件文、繰り返し文の認められるネスティングは最大 32 層です。

6.2 ラベル付き文

labeled_statement :

identifier : statement

case constant_exp : statement

default : statement

文は、ラベル前置子を持つことができます。識別子からなるラベルは、その識別子の宣言です。識別子ラベルは、goto 文の飛び先としてだけ使用されます。

識別子は、現在の関数の中でだけ有効です。ラベル名は、他の宣言の中にある同じ名前の識別子とは競合しません(ローカルでもグローバルでも同じです)。これは、CCU8 がラベル専用の異なる名前空間を使用するためです。

キーワード **case** とそれに続く定数式で構成するラベルは、case ラベルです。キーワード **default** で構成するラベルを default ラベルと呼びます。case ラベルと default ラベルは、**switch** 文で使用します。この他の場所で使用すると、CCU8 のエラーが表示されます。case ラベルの定数式は、整数型のみ認められています。case ラベルと default ラベルの詳細は、**switch** 文の説明を参照してください。

ラベル自体が制御の流れを変更することはありません。

6.3 式文

expression_statement :

expression ;

;

すべての有効な式は、セミコロンで区切ることによって、文として使用できます。C 言語の式については、第 5 章を参照してください。

式文の多くは、代入か関数呼び出しです。式からのすべての副作用は次の文を実行する前に解決します。

式がない場合は、その構文をヌル文と呼びます。ヌル文は、言語の文法がその文を必要としている場合に、ヌル演算を可能にするために使用しますが、プログラムが実際にする作業は含まれていません。

do、**for**、**if**、**while** 文は、実行可能な文をこれらの文の本体として持つ必要があります。ヌル文は、実際には文の本体を必要としない場合に、構文上の要求を満たすことができます。

次に、式文の例を示します。

例 6.1

```
int x, y, z, i ;
x = y + z ;      /* x は、y + z の値が代入されます。*/
i ++ ;           /* i は、インクリメントされます。*/
```

例 6.2

```
int i, table [100] ;
for (i = 0 ; i < 100; table[i++] = 0)
       ;
```

この例では、table[i++] = 0 のループ式が、配列 table の先頭から 100 個の要素を 0 に初期化しています。これ以上の文は必要ないので、文の本体は、ヌル文です。

6.4 複文

compound_statement :

{ declaration_list statement_list }

{ declaration_list }

{ statement_list }

{ }

declaration_list :

declaration

declaration_list declaration

statement_list :

statement

statement_list statement

複文は、ブロックとも呼ばれます。複数の文を 1 つの文として使用できるようにするために、複文が用意されています。

複文は、文のリストが続く任意の宣言を持ちます(文のリストも任意指定です)。すべて中括弧で囲みます。宣言が含まれている場合、宣言されている変数は、そのブロックのローカル変数であり、それ以降のブロックに対しては、同じ名前での他の変数の宣言を無効にします。外部宣言はブロックの終わりで有効になります。

ブロックの中に含まれる自動オブジェクトの初期化は、そのブロックに先頭から入るごとに実行されます。ブロック内での静的オブジェクトの初期化が行われるのは一度だけです。

例 6.3

```
int y, z ;
fn ()
{
   int x = 10 ;
   z = 1 ;
   if (x > y)
      x++ ;
   else
      y++ ;
}
```

6.5 選択文

選択文は、指定された条件を検査し、その結果に基づいて、制御のいくつかの流れの 1 つを選択します。2 種類の選択文があります。

1. if 文
2. switch 文

6.5.1 if 文

selection_statement：

if (expression) statement

if (expression) statement1 else statement2

「**if**」文の「**else**」の部分は省略することができます。「**if**」文はキーワード「**if**」に続けて、必ず、括弧の中に式を 1 つ指定しなければなりません。式として使用できる型は、算術型またはポインタ型です。式は、すべての副作用を含めて評価されます。

式(expression)の結果が非ゼロの場合、文 1(statemenet1)が実行されます。式がゼロの場合は、文 1 は実行されず、文 2(statement2)がある場合は、実行されます。

「**if**」文が「**else**」句にネストされている場合は、「**else**」句は、「**else**」句を持たない直前の「**if**」文の条件に一致します。

例 6.4

```
int i, j, x ;
if (i < j)
    function (i) ;
else
{
    i = x ++ ;
    function (i) ;
}
```

6.5.2 switch 文

selection_statement :

switch (expression) statement

「**switch**」文は、本体内の文に制御を移します。制御は、**switch** 式(expression)の値と一致する **case** 定数式のある文に移ります。**switch** 文には、任意の数だけ **case** を含むことができます。文(statement)の本体の実行は、選択された文の先頭から開始し、本体の終了まで進みます。また、「**break**」文が出現した時点で終了します。

「**default**」文は、**switch** 式の値に一致する **case** 定数式が存在しない場合に実行されます。「**default**」文が記述されていないのに、一致する **case** が見つからない場合は、**switch** の本体中の文は、1 つも実行されません。「**default**」文は、最後に記述しなければならないという制約はありません。「**switch**」文の本体中であれば、任意の場所に記述することができます。

switch 式の型として認められているのは整数です。各 **case** 定数式は、通常の算術変換によって変換されます(第 5.3.2 章を参照してください)。各 **case** 定数式の値は、文の本体の中では、一意でなければなりません。

switch 文の本体の **case** および **default** ラベルは、文の本体のどこから実行を開始するのかを決める最初の検査においてのみ重要な役割を果たします。実行の開始から終了までの本体に記述されているすべての文は、その文の持つラベルとは無関係にすべて実行されます。ただし、文が制御を完全に本体の外部へ移す場合を除きます。

次の例は、3 つの LED 表示項目の中から 1 つを選択するための switch 文です。

例 6.5

```
display_fn ()
{
/*このプログラムは、入力に基づいて表示の 3 種類を表示します。*/
int display_item ;
    while ( ( display_item = get_display_item () ) )
    {
       switch ( display_item )
       {
           case DATE : display_date () ;
           break ;
           case DAY : display_day () ;
           break ;
           case TIME : display_time () ;
           break ;
       }
    }
}
```

Switch文での宣言

宣言は、switch の本体を構成する複文の先頭に現れます。しかし、これらの宣言に含まれている初期化は、実行されません。switch 文は、本体内の実行可能な文に直接に制御を移すため、初期化に関する記述を含む行を無視するためです。

例 6.6

```
int y ;
switch (character)
{
    int x = 1 ;     /* 誤った初期化です。*/
    case ‘a’ :
           {
              int x = 10 ;       /* 正しい初期化です。*/
              y = x ;
              break ;
           }
    case ‘b’ :
    .
    .
}
```

6.6 繰り返し文

これ以降で説明する文は、式の評価が false になるまで繰り返し実行されます。

6.6.1 for 文

iteration_statement :

for ([expression1] ; [expression2] ; [expression3]) statement

「**for**」文は、3 つの式を評価し、文(expression)(ループ本体)を実行します。式 2(expression2)の評価が false になったら、ループを終了します。「**for**」文は、特定の回数だけループ本体を実行するというような場合に使用します。

「**for**」文は、ループ本体をゼロ回または何度も実行します。それは、3 種類の任意の制御式を使用します。「**for**」文は、次のステップを実行します。

1. 任意な式 1(expression1)を、ループに入る前に一度だけ評価します。通常は、変数の初期値が指定されています。
2. 任意な式 2(expression2)は、ループを実行する前に毎回評価されます。式の評価が false になったら、「**for**」ループ本体の実行を終了します。式の評価が true の場合は、ループ本体を実行します。
3. 任意な式 3(expression3)はループの各回の実行後、評価されます。通常は、式 1 で初期化された変数の最初期値を指定します。

4. 式 2 が false になるまで、または、**break**、**goto**、**return** などの文によって中断されるまで、「**for**」文のループを続けます。

例 6.7

```
int i ;
char string1 [20], string2 [20] ;
for (i = 0; i < 15; i++)
     string1 [i] = string2 [i] ;
```

上記の例は、string2 の先頭の 15 文字を string1 に複写します。

次の「for」文は無限ループになります。

例 6.8

```
int i, j ;
for ( ;; )
{
       j = i + 10 ;
}
```

無限ループは、**goto**、**break**、**return** 文で抜けることができます。

6.6.2 while 文

iteration_statement :

while (expression) statement

「**while**」文は、式(expression)を評価して、式の評価が false になるまで、文(expression)(ループ本体)をゼロ回または何回も実行します。

括弧の中の式を初めて評価したときに false になった場合は、ループ本体は一度も実行されません。

例 6.9

```
int x, array [15] ;
fn ()
{
   x = 0 ;
   while (x < 10)
   {
      array [x] = x ;
      x++ ;
   }
}
```

上記の例では、配列の先頭から 10 個の要素に 0 から 9 の値が代入されます。

6.6.3 do 文

iteration_statement :

do statement while (expression) ;

「**do**」文は、while 句の式(expression)の評価が false になるまで、文(expression)(ループ本体)を繰り返し実行します。

文は、少なくとも一度は実行されます。ループ本体の実行を終えるたびに式が評価されます。式が true であれば、再び文が実行されます。

例 6.10

```
int num ;
do
{
   num = get_number () ;
} while (num <= 100) ;
```

上記の例では、100 より大きくなるまで、数字を取得します。

6.7 ジャンプ文

ジャンプ文は、制御を無条件に移すものです。次の文は、ジャンプ文です。

1. goto 文
2. break 文
3. continue 文
4. return 文

「**goto**」文以外の文は、別の文の実行を中断するために使用されます。これらの文は、基本的には「switch」文とループを終了するために使用します。

6.7.1 goto 文

jump_statement :

goto identifier ;

「**goto**」文は、制御をラベル付けされている文に自動的に移します。ラベル識別子は、goto 文を含んでいる関数の有効範囲の中に記述されていなければなりません。

C 言語の他の文と同様に、複文の中のあらゆる文は、ラベルを利用することができます。**goto** 文は、複文の中に移っていくことができます。しかし、変数を初期化する宣言を持つ複文の場合は、中に移るのは危険です。複文の場合では、宣言は実行可能な文の前に現れるので、複文の中の実行可能な文に直接に移れば、初期化を無視することになるからです。この場合、結果は定義されません。

次に、「**goto**」文とラベル付けされている文の使用例を示します。

例 6.11

```
int error_no ;
fn ()
{
   extern int error ;
   if (error )
      goto error_process ;
   .
   error_process :
   return (error_no) ;
}
```

上記の例では、「**goto**」文が制御を error_process というラベル付けされている場所へ移ります。

6.7.2 break 文

jump_statement :

break ;

break 文は、直前の括弧で囲まれている **while**、**do**、**for**、**switch** 文を終了させます。制御は、ループ本体に従って文に移ります。

例 6.12

```
int x ;
while (1)
{
   x = fn () ;
   if (x == 1)
      break ;
}
```

この例では、関数が値 1 を返すまで、while ループが実行されます。

6.7.3 continue 文

jump_statement :

continue ;

continue 文は、制御を、直前の括弧で囲まれている **while**、**do**、**for** 文の終わりに移します。制御は、次の繰り返し文の **while**、**do**、**for** 文に移ります。このとき、ループ本体の残りの文を無視します。

例 6.13

```
even_fn ()
{
   int x ;
   for (x = 0; x <= 100; x++)
   {
      if (x%2)
        continue ;
      print (x) ;
   }
}
```

上記の関数は、0 から 100 までのすべての偶数をプリントアウトします。

6.7.4 return 文

jump_statement :

return [expression] ;

return 文は、戻り値の有無とは無関係に、関数から戻ります。

CCU8 は、式が指定されている場合は、これを評価し、呼び出し元の関数に値を返します。必要な場合は、コンパイラは関数で宣言されている型へ値を変換します。指定されている戻り値がない場合は、値は定義されません。

例 6.14

```
max (int a, int b)
{
  if (a > b)
    return (a) ;
  else
    return (b) ;
}
```

上記の関数は、2 つの整数型の引数のうち、大きい方を返します。

戻り値を何も持たない(void)ことを宣言されている関数が値を返すと、コンパイラはエラーメッセージを出力します。

6.8 ASM 文

asm_statement :

__asm (“string”)

asm 文 を使用すると、アセンブリ出力ファイルにある文字列の内容を直接記述することができます。文字列(string)がコンパイラによって処理されることはありません。

例 6.15

INPUT:

```
__asm (“; Test for __asm”)
int gvar ;
fn (int arg)
{
```

```
            gvar = arg ;
            __asm (“\tl   er0,   NEAR _gvar\n”) ;

      }
```

OUTPUT :

```
CFILE 0000H 0000000AH "test.c"
; Test for __asm
      rseg $$NCODtest
_fn   :
;;gvar = arg ;
      st    er0,   NEAR _gvar
;;__asm ("\tler0,   NEAR _gvar\n") ;
      l     er0,   NEAR _gvar
;;}
      rt
      public _fn
      _gvar comm data 02h #00h
      extrn code near : _main

            end
```

7. ANSI C 標準との相違点

CCU8 の C 言語の実装では、以下の機能をサポートしているため、ANSI (American National Standards Institute)が提案した標準の ANSI X3. 159-1989 および ISO/IEC 9899 と異なる部分があります。

1. INTERRUPT 関数の仕様をサポートしています。
2. 構造体または共用体のメンバーを、「**const**」で修飾することはできません。
3. 構造体および共用体で、**char** のビットフィールドを使用できます。
4. 引数を「**const**」で修飾することはできません。
5. 型指定子、型修飾子、または記憶域クラス指定子がない宣言は、「**int**」型の宣言として扱われます。
6. 関係演算子を使用して、ポインタを定数の整数式と比較したり、void へのポインタと比較することができます。
7. **__near** および**__far** のメモリモデル修飾子がサポートされています。
8. **__noreg** 関数修飾子がサポートされています。
9. **__asm** キーワードがサポートされています。
10. シングルラインコメント(//)がサポートされています。
11. 次の組み込み関数がサポートされています。

```
__EI    __DI    __segbase_n    __segbase_f    __segsize
```