RX ファミリ用 C/C++ コンパイラパッケージ

アプリケーションノート：＜コンパイラ活用ガイド＞

言語編(C89,C99)

RJJ06J0095-0100
Rev.1.00
2010.04.20

本ドキュメントでは、RX ファミリ用 C/C++コンパイラ V.1.0 で新規に追加された C99 言語仕様を紹介します。

目次

1. はじめに

RX ファミリ用 C/C++コンパイラ V.1.0 では、ANSI 準拠 C 言語 C89、ANSI 準拠 C++言語、EC++言語のほかに、C89 の改訂版である ANSI 準拠 C 言語 C99 (可変長配列は除く)を使用することができます。本書では、C99 での新機能と、C89 から C99 へ移行する際の注意事項を説明します。

RX ファミリ用 C/C++コンパイラは、デフォルト（コンパイルオプション-lang=c）ではソースプログラムを C89 でコンパイルします。コマンドラインで C99 向けにコンパイルするには、コンパイルオプション -lang=c99 を指定してください。High-performance Embedded Workshop では、[ビルド → RX Standard Toolchain] メニューを選択して表示される[RX Standard Toolchain]ダイアログで設定します。

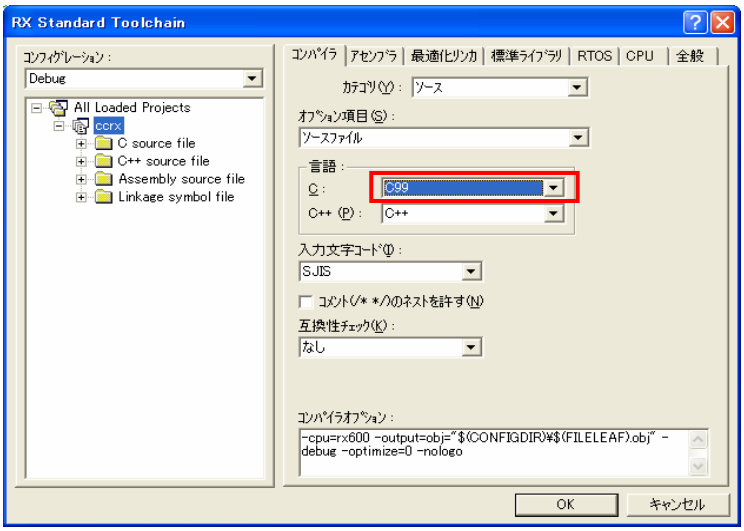

図 1-1 C99 用コンパイラオプション

また、C99 で新規に追加された標準ライブラリの中には、標準ライブラリ構築ツールの設定が必要なライブラリもあります。コマンドラインで使用する場合には、-lang=c99 オプションと-headオプションを使用してください。High-performance Embedded Workshopでは、[RX Standard Toolchain]ダイアログで設定します。詳しくは「2.5 標準インクルードファイル」を参照してください。

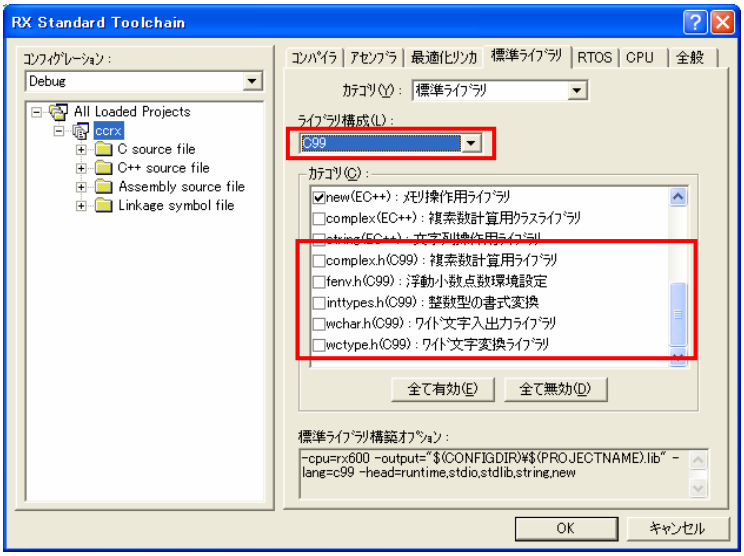

図 1-2 C99 用標準ライブラリ構築ツールオプション

2. C99 新規機能

2.1 プリミティブ型

C99 では、プリミティブ型として、論理型、複素数型、long long 型、可変長配列型、フレキシブル配列メンバが追加されました。

2.1.1 論理型

■ポイント

真か偽かを表すための論理型が追加されました。

■説明

C99 では、真か偽かを表す論理型として、bool 型、_Bool 型が追加されました。また、bool 型（_Bool 型）の値として、真を表す true と、偽を表す false が使用できます。bool 型、_Bool 型、true、false を使用するには、stdbool.h をインクルードしてください。

■使用例

(1) bool 型、true、false の使用

bool 型変数を使用した例です。

```
#include <stdbool.h>

int func(bool b, bool c)
{
    if (b == true) {
        return 1;
    }

    if (c == false) {
        return -1;
    }

    return 0;
}
```

(2) 判定式

int 型の変数を比較した結果を bool 型で返します。

```
#include <stdbool.h>

bool func(int a, int b)
{
    return (a==b);
}
```

2.1.2　複素数型

■ポイント

複素数を扱うための_Complex 型と、複素数の虚数部を扱うための_Imaginary 型が追加されました。

■説明

C99 では、複素数を扱うための型が追加されました。複素数は、実数部と虚数部からなりたっています。複素数型として、float _Complex 型、double _Complex 型、long double _Complex 型の３種類の型が使用できます。例えば float _Complex 型は、実数部と虚数部がそれぞれ float 型であるような複素数型であることを意味します。また、複素数の虚数部を扱うために、float _Imaginary 型、double _Imaginary 型、long double _Imaginary 型の３種類の型が使用できます。例えば float _Imaginary 型は、float 型の虚数であることを意味します。さらに、虚数単位（数学的には i^2=-1 になるような i と定義されます）を扱うために__I__が使用できます。なお、”_Complex”の代わりに”complex”、”_Imaginary”の代わりに”imaginary”、”__I__”の代わりに”I”を使用することもできます。複素数型を使用するには、complex.h をインクルードしてください。

複素数計算ライブラリを使用するには、標準ライブラリ構築ツールでオプション-head=complex を指定する必要があります。

■使用例

(1) 複素数型の使用

```
#include <complex.h>

float _Complex cf;
double _Complex cd;
long double _Complex cld;

void func()
{
    /* 実数部が1.0、虚数部が2.0のfloat型の複素数 */
    cf = 1.0f + __I__ * 2.0f;

    /* 実数部が10.0、虚数部が20.0のdouble型の複素数 */
    cd = 10.0 + __I__ * 20.0;

    /* 実数部が100.0、虚数部が200.0のlong double型の複素数 */
    cld = 100.0 + __I__ * 200.0;
}
```

(2) 複素数型の演算

```
#include <complex.h>

float _Complex cf1, cf2, cf3;

void func1()
{
    /* 複素数同士の加算 */
    cf1 = cf2 + cf3;
}

void func2()
{
    /* 複素数同士の減算 */
    cf1 = cf2 - cf3;
}

void func3()
{
    /* 複素数同士の乗算 */
    cf1 = cf2 * cf3;
```

```
}

void func4()
{
    /* 複素数同士の除算 */
    cf1 = cf2 / cf3;
}

int func5()
{
    /* 複素数同士の比較 */
    if (cf1 == cf2) return 1;
    else if (cf1 != cf2) return -1;
}
```

(3) 型変換

```
#include <complex.h>

float _Complex cf;
double _Complex cd;
float fr;
float _Imaginary fi;

void func1()
{
    /* float _Complex → floatへの型変換。実数部を取得できます。 */
    fr = (float)cf;
}

void func2()
{
    /* float _Complex → float _Imaginaryへの型変換。虚数部を取得できます。 */
    fi = (float _Imaginary)cf;
}

void func3()
{
    /* float → float _Complexへの型変換。cfの虚数部は0になります。 */
    cf = (float _Complex)fr;
}

void func4()
{
    /* float _Imaginary → float _Complexへの型変換。cfの実数部は0になります。 */
    cf = (float _Complex)fi;
}

void func5()
{
    /* float _Complex → double _Complexへの型変換。
    ** 実数部と虚数部がそれぞれfloatからdouble型に変換されます。
    */
    df = (double _Complex)cf;
}
```

(4) 虚数単位

```
#include <complex.h>

float _Complex cf1, cf2;

Void func()
{
    /* cf2の実数部がcf1の虚数部に、cf2の（虚数部×-1）がcf1の実数部に代入されます。 */
    cf1 = cf2 * __I__;

    /* 実数部は-1に、虚数部は0になります。 */
    cf2 = __I__ * __I__;
}
```

2.1.3 long long 型

■ポイント

64 ビットの整数型を扱うために、long long 型が追加されました。

■説明

C99 では、64 ビットの整数型を扱うために、long long 型（signed long long 型、unsigned long long 型）が言語仕様として正式に規定されました。

なお RX ファミリ C/C++コンパイラでは、C89 でも long long 型が使用できます。

long long 型で扱える値の範囲は以下の通りです。

表 2-1 long long 型の値範囲

型	最小値	最大値
signed long long 型	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long 型	0	18,446,744,073,709,551,615

long long 型の整数定数を表すための接尾語が追加されています。

表 2-2 long long 型定数の接尾語

整数定数の型	接尾語
signed long long 型	LL,ll
unsigned long long 型	ULL,LLU,ull,llu

printf や sprintf などで long long 型を扱うために、%llx、%llX、%lld、%llu などの書式が追加されています。

表 2-3 long long 型の書式

型	書式
signed long long 型	%lld（signed long long 型の値を 10 進数で書式変更） %llx（signed long long 型の値を 16 進数で書式変更）
unsigned long long 型	%llu（unsigned long long 型の値を 10 進数で書式変更） %llX（unsigned long long 型の値を 16 進数で書式変更）

■使用例

long long 型を使用した例です。

```
#include <stdio.h>   /* for printf */

long long sll;
unsigned long long ull;

void func1()
{
    sll = 0x7FFFFFFFFFFFFFFFLL;    /* signed long long型の整数定数 */
    ull = 0xFFFFFFFFFFFFFFFFULL;   /* unsigned long long型の整数定数 */
}

void func2()
{
    printf("%lld¥n", sll);    /* signed long long型の変数の値を10進数で出力 */
    printf("%llx¥n", sll);    /* signed long long型の変数の値を16進数で出力 */

    printf("%llu¥n", ull);    /* unsigned long long型の変数の値を10進数で出力 */
    printf("%llX¥n", ull);    /* unsigned long long型の変数の値を16進数で出力 */
}
```

2.1.4　可変長配列型

■ポイント

C99 では可変長配列型が使用できるようになりました。

ただし、RX ファミリ C/C++コンパイラでは可変長配列型はサポートされていません。

2.1.5　フレキシブル配列メンバ

■ポイント

構造体や共用体の最後のメンバに、要素数を指定しない配列メンバ（フレキシブル配列メンバ）を宣言できるようになりました。

■説明

C89 では、配列型の構造体・共用体メンバを宣言する際、要素数の指定が必要でした。C99 では、構造体や共用体の最後のメンバに、要素数を指定しない配列メンバ（フレキシブル配列メンバ）を宣言できます。

ビットマップなどの画像データフォーマットや、TCP/IP などの通信データフォーマットで扱う構造体では、構造体の冒頭にデータサイズなどのヘッダ情報が宣言され、ヘッダ情報の後にデータ部分が配列型で宣言されることがあります。さらに、データ部分のサイズは実行時でなければ分からないことがあります。フレキシブル配列メンバにより、このようなデータフォーマットを容易に扱うことができます。

■使用例

構造体 S のメンバ data がフレキシブル配列メンバです。C89 の場合、構造体 S にダミーの data[1]を宣言する必要があります。このため、構造体 S 用のデータを確保する際、sizeof(struct S)のサイズから、data[1]のサイズを減算しています。

C89

```
#include <stdlib.h>    /* for malloc */

struct S {
    int size;
    int dummy1;
    int dummy2;
    int data[1];
};

void func(int size)
{
    struct S *p;
    int i;

    p = (struct S*)malloc(
              sizeof(struct S)
              - sizeof(int)
              + sizeof(int) * size);

    p->size = size;
    for (i=0; i<size; i++) {
        p->data[i] = 0;
    }
    ...
}
```

C99

```
#include <stdlib.h>    /* for malloc */

struct S {
    int size;
    int dummy1;
    int dummy2;
    int data[];    /* フレキシブル配列メンバ */
};

void func(int size)
{
    struct S *p;
    int i;

    p = (struct S*)malloc(
              sizeof(struct S)

              + sizeof(int) * size);

    p->size = size;
    for (i=0; i<size; i++) {
       p->data[i] = 0;
    }
    ...
}
```

■注意

(1) sizeof 演算子

フレキシブル配列メンバを含む構造体・共用体に対する sizeof 演算子の戻り値には、フレキシブル配列メンバのサイズは含まれません。下記の例では、data[]を除く構造体サイズである 12 が x に格納されます。

```
void func()
{
    int x = sizeof(struct S);
}
```

(2) フレキシブル配列メンバを含む構造体・共用体型の変数

フレキシブル配列メンバを含む構造体・共用体型の変数（フレキシブル配列メンバを含む構造体・共用体へのポインタ型はこれに含まれません）が宣言されている場合に、その変数のフレキシブル配列メンバへのアクセスは保証されません。下記の例では、構造体 S 型の変数である s は size、dummy1、dummy2 の領域しか持たないため、s.data[0]へのアクセスは不正アクセスになります。

```
struct S s;
void func()
{
    s.data[0] = 0;
}
```

(3) フレキシブル配列メンバを含む構造体・共用体のコピー

フレキシブル配列メンバを含む構造体・共用体をコピーする場合、フレキシブル配列メンバはコピーされません。下記の例では、(*p)から(*q)へのコピーにより size、dummy1、dummy2 はコピーされますが、data[]の領域はコピーされません。

```
void func(struct S *p, struct S *q)
{
    *q = *p;
}
```

2.2 キーワード

C99 では、キーワード inline、restrict、_Pragma が追加されました。

2.2.1 inline

■ポイント

関数をインライン展開するようにコンパイラに指示を与えるために、inline キーワードがサポートされました。

■説明

関数に inline キーワードを指定することにより、コンパイラに対して関数のインライン展開を指示できるようになりました。ただし inline キーワードは、コンパイラがインライン展開を実施するためのヒントを与えるだけに過ぎず、実際にインライン展開を実施するかどうかは、コンパイラの処理に依存します。

■使用例

inline を使用して、関数 add のインライン展開をコンパイラに指示します。

ソースコード記述

```
static inline int add(int x, int y)
{
    return (x + y);
}

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
    yyy = add(yyy, 2);
}
```

コンパイラの解釈（-inline=100オプション指定時）

```
int xxx, yyy, zzz;
void func()
{
    xxx = yyy + zzz;
    yyy = yyy + 2;
}
```

■注意

外部リンケージを持つ関数（extern 宣言された関数）に対して inline 指定された関数宣言を記述した場合には、同じソースファイル内にその関数の定義がなければなりません。関数定義がないと、リンク時に未定義シンボルエラー（L2310）が出力されることがあります。inline 指定した関数は、同じソースファイル内に必ず関数を定義してください。

```
inline int add(int x, int y);   /* 宣言のみで定義がない */

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
}
```

関数が inline 指定されていても、コンパイラの条件によりインライン展開されない場合があります。このとき、inline 指定された関数が extern 宣言されていないと、関数定義が生成されません。その結果、リンク時に未定義シンボルエラー（L2310）が出力されることがあります。inline 指定された関数に対して L2310 が出力される場合には、当該関数に extern 宣言を追加してください。例えば下記の例の場合、コンパイルオプション-inline=0 を指定した場合には、関数 add はインライン展開されず、また add の定義も生成されないため、リンク時に L2310 が出力されます。リンクエラーを回避するには、関数 add を extern 宣言します。

修正前

```
inline int add(int x, int y)
{
    return (x + y);
}

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
}
```

修正後

```
inline extern int add(int x, int y)
{
    return (x + y);
}

int xxx, yyy, zzz;
void func()
{
    xxx = add(yyy, zzz);
}
```

なお、RX ファミリ用 C/C++コンパイラでは、#pragma 拡張機能を使用したインライン展開（#pragma inline）を利用することができます。C99 の inline キーワードを使用した場合は、上記の通り、inline 指定された関数が extern 宣言されていないときにその関数の定義が生成されません。一方、#pragma inline を使用した場合は、inline 展開できたかどうかに関わらず関数定義が生成されます。

inline 指定された関数には、内部リンケージを持つ識別子（関数内 static 変数やファイル内 static 変数）への参照を含むことはできません。

```
inline void add1()
{
    /* C6030エラーになります */
    static int x = 0;
    x ++;
}

static int y;
inline void add2()
{
    /* C6031エラーになります。 */
    x ++;
}

void func()
{
    add1();
    add2();
}
```

2.2.2 restrict

■ポイント

コンパイラにポインタ向け最適化のヒントを与えるために、restrict キーワードが追加されました。

■説明

コンパイラに対し、restrict 修飾されたポインタが示す領域と、他のポインタが示す領域が重複しないことを明示することにより、コンパイラがポインタ向けの最適化を実施しやすくなります。

■使用例

(1) restrict を使用しない場合

通常コンパイラは、(*q)や(*r)の領域と(*p)の領域は重なる可能性があるとみなします。このため、(*p)へのストアにより(*q)や(*r)が書き換わる可能性があるとみなされ、ループ内部で(*q)や(*r)を毎回ロードするコードが生成されます。

ソースプログラム：

```
void func(int * p,
          int * q, int * r,
          int n)
{
   int i;

   for (i=0; i<n; i++) {
      p[i] = *q + *r;
   }
}



int a[10], b, c;
void main()
{
   func(a, &b, &c, 10);
}
```

生成コード：

```
_func:
   MOV.L      R4,R15
   MOV.L      #00000000H,R4
   BRA        L11
L12:
   MOV.L      [R2],R5    ; (*q)のロード
   ADD        #01H,R4
   ADD        [R3],R5    ; (*r)のロード
   MOV.L      R5,[R1+]   ; (*p)へのストア
L11:
   CMP        R15,R4
   BLT        L12
L13:
   RTS
```

(2) restrict を使用した場合

q や r を restrict 修飾することにより、 (*q)や(*r)の領域が(*p)の領域と重ならないことをコンパイラに対して明示的に指示できます。これにより、コンパイラは(*p)へのストアがあっても(*q)や(*r)は書き換わらないとみなすことができ、ループ先頭のただ 1 回だけ(*q)や(*r)をロードするような最適化されたコードを生成できます。

ソースプログラム：

```
void func(int * p,
          int * restrict q, int * restrict r,
          int n)
{
   int i;

   for (i=0; i<n; i++) {
      p[i] = *q + *r;
   }
}



int a[10], b, c;
void main()
{
   func(a, &b, &c, 10);
}
```

生成コード：

```
_func:
   MOV.L      R4,R15
   MOV.L      [R2],R4    ; (*q)のロード
   MOV.L      #00000000H,R5
   ADD        [R3],R4    ; (*r)のロード
   BRA        L11
L12:
   MOV.L      R4,[R1+]   ; (*p)へのストア
   ADD        #01H,R5
L11:
   CMP        R15,R5
   BLT        L12
L13:
   RTS
```

2.2.3 _Pragma

■ポイント

#pragma と同じ機能を提供する演算子として、_Pragma キーワードが追加されました。

■説明

#pragma は処理系により使用方法が異なるため、使用するコンパイラの仕様に合わせて#pragma の書き方を変更する必要があります。ソースプログラム内で#pragma を頻繁に使用すると、#pragma 記述のたびにコンパイラ種別に応じて#pragma の記述を切り分ける必要があるため、ソースプログラムの可読性が悪くなることがあります。C99 で追加された_Pragma キーワードを使用することにより、より簡潔に#pragma を記述できます。

■使用例

(1) 最適化方針の切り替え

単一ファイル内の任意の位置に宣言することにより関数単位に最適化方針を切り替えられる#pragma があると仮定します。下記の例では、関数 func_001 を最適化ありに、関数 func_002 を最適化なしに設定しています。C89 では、func_001 と func_002 の定義の前に、それぞれコンパイラ種別の判定と#pragma 記述が必要でした。C99 では、_Pragma を使用することにより、より簡潔に記述できます。

C89

```
#ifdef __RX
/* RXコンパイラ用の宣言(最適化あり) */
#pragma option optimize=2
#else
/* 他のコンパイラ用の宣言(最適化あり) */
#pragma ...
#endif




int x;

void func_001()
{
    x ++;
    x ++;
}

#ifdef __RX
/* RXコンパイラ用の宣言(最適化なし) */
#pragma option optimize=0
#else
/* 他のコンパイラ用の宣言(最適化なし) */
#pragma ...
#endif

void func_002()
{
    x ++;
    x ++;
}
```

C99

```
#ifdef __RX
/* RXコンパイラ用の宣言 */
#define OPTIMIZE_ON  _Pragma("option optimize=2")
#define OPTIMIZE_OFF _Pragma("option optimize=0")
#else
/* 他のコンパイラ用の宣言 */
#define OPTIMIZE_ON  _Pragma("...")
#define OPTIMIZE_OFF _Pragma("...")
#endif



int x;

OPTIMIZE_ON void func_001()
{
    x ++;
    x ++;
}









OPTIMIZE_OFF void func_002()
{
    x ++;
    x ++;
}
```

(2) エンディアンの切り替え

単一ファイル内の任意の位置に宣言することにより変数単位にエンディアンを切り替えられる#pragma があると仮定します。下記の例では、変数 x_little をリトルエンディアンに、変数 x_big をビッグエンディアンに設定しています。C89 では、x_little と x_big の定義位置の前に、それぞれコンパイラ種別の判定と#pragma 記述が必要でした。C99 では、_Pragma を使用することにより、より簡潔に記述できます。

C89

```
#ifdef __RX
/* RXコンパイラ用の宣言(little endian) */
#pragma endian little
#else
/* 他のコンパイラ用の宣言(little endian) */
#pragma ...
#endif



int x_little;

#ifdef __RX
/* RXコンパイラ用の宣言(big endian) */
#pragma endian big
#else
/* 他のコンパイラ用の宣言(big endian) */
#pragma ...
#endif

int x_big;
```

C99

```
#ifdef __RX
/* RXコンパイラ用の宣言 */
#define ENDIAN_LITTLE   _Pragma("endian little")
#define ENDIAN_BIG      _Pragma("endian big")
#else
/* 他のコンパイラ用の宣言 */
#define ENDIAN_LITTLE   _Pragma("...")
#define ENDIAN_BIG      _Pragma("...")
#endif

ENDIAN_LITTLE int x_little;









ENDIAN_BIG int x_big;
```

2.3 リテラル

C99 で新たに使用できるようになったリテラルについて説明します。

2.3.1 浮動小数点の 16 進数表記

■ポイント

浮動小数点の定数値を 16 進数で表記できるようになりました。

■説明

C99 では、浮動小数点数を 16 進数で表すことができます。これにより、10 進数で表すよりも誤差を小さくできます。例えば 0.1 という 10 進数の浮動小数点数を記述したとします。0.1 は 16 進数で正確に表すことができないため、コンパイラが 16 進数で表現する際、丸めや誤差などの影響により値が変化することがあります。しかし、16 進数で浮動小数点数を記述すると、丸めや誤差などの影響を受けずに値を固定できます。浮動小数点数の 16 進数表記の書式は以下の通りです。

0xaaaa.bbbbPdd　　(P は小文字も可)

先頭から”0x”、整数部 aaaa、小数点”.”、及び小数部 bbbb の順に書きます。P 以下は正または負の 10 進数で指数を記述します。なお、小数点や小数部は省略できますが、P 以下は省略できません。

■使用例

浮動小数点数 0.1 を 10 進数で記述した場合、コンパイルオプション-round により、16 進数での値が変化します。しかし、16 進数で記述した場合には、オプションの指定に関わらず値を固定できます。

浮動小数点数を10進数で記述した場合	浮動小数点数を16進数で記述した場合
`float x = 0.1f;`	`float x = 0x0.CCCCCCP-3;`
オプションによる値の変化	オプションによる値の変化
`-round=zero    : 0x3DCCCCCC` `-round=nearest  : 0x3DCCCCCD`	`-round=zero    : 0x3DCCCCCC` `-round=nearest  : 0x3DCCCCCC`

2.3.2 enum

■ポイント

列挙型の最後の要素の後ろにカンマがあっても、正常にコンパイルされるようになりました。

■説明

C89 では、enum 宣言の最後に余分なカンマがあった場合、エラーになる仕様でした。しかし、C99 ではエラーにならないよう、仕様が変更されています。

なお、RX ファミリ用 C/C++コンパイラでは、C89 であっても余分なカンマの使用は許されます。

■使用例

enum 宣言の最後である CCC の後にカンマがあってもコンパイルエラーになりません。

```
enum E { AAA, BBB, CCC, };

enum E e = CCC;
```

2.3.3　配列・構造体の初期化

■ポイント

配列や構造体を初期化する際、特定の要素やメンバを指定して初期化できるようになりました。

■説明

C99 では、配列の要素番号及び構造体のメンバを明示的に指定できる、指定初期化子という形式が使用できるようになりました。配列を初期化する際、C89 では各要素を先頭から順番に初期化する必要があります。しかし、C99 では特定の要素にのみ初期値を設定することができます。このとき、初期化していない要素は 0 で初期化されます。構造体のメンバも同様、C89 では先頭のメンバから順番に初期化する必要があります。C99 では、特定のメンバにのみ初期値を設定することができます。このとき、初期化していないメンバは 0 で初期化されます。初期化が必要な要素やメンバが限定されている場合、あるいは、要素数が大きな配列や多くのメンバを持つ構造体のうち特定の要素・メンバのみ初期化が必要な場合などに便利な機能です。

■使用例

(1) 配列での使用例

array の 3 番目と 4 番目の要素に初期値を指定します。それ以外は値が 0 になります。

C89

```
int array[5] = { 0, 0, 0, 2, 1 };
```

C99

```
int array[5] = { [3] = 2, [4] = 1 };
```

(2) 構造体での使用例

特定の構造体メンバのみ初期値を設定します。構造体 T のように、構造体がネストしている場合でも使用できます。構造体 S1 のように、間に多くのメンバを含む場合であっても、特定のメンバだけ初期化できます。

C89

```
struct S {
    int a;
    int b;
} s = { 0, 1 };

struct T {
    int a;
    int b;
    struct T1 {
        int aa;
        int bb;
    } t1;
} t = { 0, 1, { 0, 2 } };

struct S1 {
    int a;
    int b[100];
    int c;
} s1 = { 10, {0,0,…}, 20 };
```

C99

```
struct S {
    int a;
    int b;
} s = { .b = 1 };

struct T {
    int a;
    int b;
    struct T1 {
        int aa;
        int bb;
    } t1;
} t = { .b = 1, .t1.bb = 2 };

struct S1 {
    int a;
    int b[100];
    int c;
} s1 = { .a = 10, .c = 20 };
```

2.3.4　複合リテラル

■ポイント

構造体や配列型の即値を記述できます。

■説明

C99 では、初期値を持つ無名のオブジェクトを作成できます。これにより、初期化された配列データや構造体データを扱う場合に、より簡潔に記述できるようになります。複合リテラルの書式は下記のようになります。

(型 [要素数]){ 要素 1, 要素 2, … }

■使用例

C89 では、Point 型の変数 temp を宣言し、メンバを初期化したうえで関数 func に渡す必要があります。しかし C99 では、temp のような一次変数を用意することなく、直接関数に値を渡すことができます。

C89

```
typedef struct Point {
    short x,y;
} Point;

Point x;

void func(Point *p)
{
    x = *p;
}

void func_1()
{
    Point temp = {100,200};
    func(&temp);
}

void func_2()
{
    Point temp[2] = {{100,200},{300,400}};
    func(temp);
}
```

C99

```
typedef struct Point {
    short x,y;
} Point;

Point x;

void func(Point *p)
{
    x = *p;
}

void func_1()
{

    func(&(Point){100,200});
}

void func_2()
{

    func((Point[2]){{100,200}, {300,400}});
}
```

2.4 文法

C99 で新たに追加された文法について説明します。

2.4.1 一行コメント

■ポイント

C++の一行コメントが使用できるようになりました。

■説明

C89 まではコメント記述は/**/のみ使用できました。C99 では、//で始まり改行で終わる C++の一行コメントも使用できるようになりました。

なお、RX ファミリ用 C/C++コンパイラでは、C89 でも//によるコメントの記述が可能です。

■使用例

一行コメントを使用した例です。

C89

```
void func()
{
    int a;
    a = 5;    /* コメント */
}
```

C99

```
void func()
{
    int a;
    a = 5;    // コメント
}
```

2.4.2　ワイド文字の連結

■ポイント

ワイド文字の連結方法が規定されました。

■説明

C 言語では、２つ以上の文字列が連続して記述されている場合、１つの文字列に連結されます。例えば、「”aaa”△“bbb”」（△は半角スペース）という文字列があった場合、「”aaabbb”」のように連結した文字列として扱われます。同様にワイド文字列が連続している場合も、「L”aaa”△L”bbb”」は「L”aaabbb”」のように連結したワイド文字列として扱われます。ここで C89 では、文字列とワイド文字列が連続している場合の連結方法は未定義動作になっていました。C99 では、文字列とワイド文字列が連続している場合、ワイド文字列として連結することが規定されました。

なお RX ファミリ用 C/C++コンパイラでは、C89 で文字列とワイド文字列を連続して記述した場合には C6282 エラーとなります。

■使用例

文字列”Application”とワイド文字列”Note”が連結され、L”ApplicationNote”と扱われます。

```
#include <wchar.h>

wchar_t *str = "Application" L"Note";
```

2.4.3　可変個引数マクロ

■ポイント

マクロに可変個引数を使用できるようになりました。

■説明

C89 では、printf や scanf などのように関数に可変個引数を使用することはできましたが、マクロでは使用できませんでした。C99 では、マクロでも可変個引数を使用できるようになりました。書式は以下の通りです。

#define　マクロ名(str,...)　可変個引数を使用する位置で__VA_ARGS__を使用

なお、マクロを使用する際、可変個引数を省略することも可能です。

■使用例

C89 では、引数の数が異なるマクロは DEBUG1、DEBUG2 のように別々に定義する必要がありました。C99 では、引数の個数が違うマクロであっても 1 つの定義で済みます。

C89

```
#include <stdio.h>

#define DEBUG1(fmt, val1)       printf("[debug] : " ## fmt, val1);
#define DEBUG2(fmt, val1, val2) printf("[debug] : " ## fmt, val1, val2);

void func()
{
    int a = 0;
    int b = 1;

    DEBUG1("a = %d¥n", a);
    DEBUG2("a = %d¥n, b = %d", a, b);
}
```

C99

```
#include <stdio.h>

#define  DEBUG(fmt, ...)  printf("[debug] : " ## fmt, __VA_ARGS__);

void func()
{
    int a = 0;
    int b = 1;

    // printf("[debug] : a = %d¥n", a); と解釈される。
    DEBUG("a = %d¥n", a);

    // printf("[debug] : a = %d, b = %d¥n", a, b); と解釈される。
    DEBUG("a = %d, b = %d¥n", a, b);

    // 下記のような記述も可能。
    // printf("[debug] : message¥n"); と解釈される。
    DEBUG("message¥n");
}
```

2.4.4　関数型マクロの空引数

■ポイント

関数型マクロに空の引数を渡せるようになりました。

■説明

C89 では、関数型マクロを使用するときに引数を省略できませんでした。C99 では、空の引数を渡すことができるようになりました。

なお RX ファミリ用 C/C++コンパイラでは、C89 でもエラーにならず、警告（C5054）が出力されるだけです。C99 と同様、空の指定と解釈されます。

■使用例

関数型マクロである MESSAGE に全ての引数を渡した場合と、一部の引数だけを渡した場合の例です。

```
#include <stdio.h>

#define MESSAGE(msg1, msg2)   "msg1 is " ## msg1 ## ". msg2 is " ## msg2 ## "."

void func()
{
    // printf("%s¥n", "msg1 is AAA. msg2 is BBB."); と解釈される。
    printf("%s¥n", MESSAGE("AAA", "BBB"));

    // printf("%s¥n", "msg1 is CCC. msg2 is ."); と解釈される。
    printf("%s¥n", MESSAGE("CCC"));
}
```

2.4.5　識別子の使用可能文字

■ポイント

識別子にユニバーサル文字や多バイト文字が使用できるようになりました。

■説明

C89 では、識別子に使用可能な文字列は、アルファベット（大文字、小文字）、数字（識別子の先頭は除く）、”_”（アンダースコア）に限定されていました。C99 では、識別子に使用可能な文字が拡張され、ユニバーサル文字や多バイト文字を使用できるようになりました。ただし、数字は識別子の先頭として使用できません。

■使用例

”文字”という名の変数名や、”関数”という名の関数名を使用する例です。

```
char ¥u6587¥u5B57;  /* "文字”という名の変数名*/

void func()
{
    ¥u6587¥u5B57 = 'a';
}

// "関数"という名の関数
void ¥u95A2¥u6570()
{
    func();
}

void main()
{
    ¥u95A2¥u6570();
}
```

2.4.6　変数の宣言位置

■ポイント

変数の宣言をブロックの途中にも記述できるようになりました。

■説明

C89 では、変数宣言はブロックの先頭に書く必要がありました。C99 からは、変数が参照される前であれば、ブロックの先頭以外にも記述できます。

■使用例

関数 func で局所変数 i、j を宣言していますが、C99 では、関数内の任意の位置で変数を宣言できます。

C89

```
int a[10], b[10];

void func()
{
    int i;
    int j;

    for (i=0; i<10; i++) {
       a[i] = 10;
    }


    for (j=0; j<10; j++) {
       b[j] = j;
    }
}
```

C99

```
int a[10], b[10];

void func()
{
    int i;


    for (i=0; i<10; i++) {
       a[i] = 10;
    }

    int j;
    for (j=0; j<10; j++) {
       b[j] = j;
    }
}
```

2.5　標準インクルードファイル

C99 では complex.h、fenv.h、inttypes.h、stdbool.h、stdint.h、tgmath.c の 6 つの標準インクルードファイルが追加されました。

2.5.1　complex.h

■ポイント

複素数計算ライブラリが追加されました。

■説明

complex.h は、複素数計算ライブラリを使用するための標準インクルードファイルです。以下に complex.h に含まれる関数の一覧を示します。float 型の複素数の場合は定義名の最後に”f”を付けた関数名、long double 型の複素数の場合は定義名の最後に”l”を付けた関数名、double 型の複素数の場合は定義名がそのまま関数名になります。

複素数計算ライブラリを使用するには、標準ライブラリ構築ツールでオプション-head=complex を指定する必要があります。

表 2-4 複素数計算ライブラリ

種別	定義名	説明
関数	cacos	複素数逆余弦を計算します。
	casin	複素数逆正弦を計算します。
	catan	複素数逆正接を計算します。
	ccos	複素数余弦を計算します。
	csin	複素数正弦を計算します。
	ctan	複素数正接を計算します。
	cacosh	複素数逆双曲線余弦を計算します。
	casinh	複素数逆双曲線正弦を計算します。
	catanh	複素数逆双曲線正接を計算します。
	ccosh	複素数双曲線余弦を計算します。
	csinh	複素数双曲線正弦を計算します。
	ctanh	複素数双曲線正接を計算します。
	cexp	複素数自然対数の底 e の z 乗を計算します。
	clog	複素数自然対数を計算します。
	cabs	複素数絶対値を計算します。
	cpow	複素数べき乗を計算します。
	csqrt	複素数平方根を計算します。
	carg	偏角を計算します。
	cimag	虚部を計算します。
	conj	虚部の符号を反転させて複素共役を計算します。
	cproj	リーマン球面上への射影を計算します。
	creal	実部を計算します。

■使用例

複素数の逆余弦を計算する例です。

```
#include <complex.h>
double complex z, ret;
void func(void)
{
    ret = cacos(z);
}
```

2.5.2 fenv.h

■ポイント

浮動小数点環境のライブラリが追加されました。

■説明

fenv.h は、浮動小数点環境へアクセスするための標準インクルードファイルです。浮動小数点環境とは、fenv.h で定義される浮動小数点状態フラグや浮動小数点例外フラグなどのことを言います。以下に、fenv.h に含まれるマクロ、関数の一覧を示します。

浮動小数点環境ライブラリを使用するには、標準ライブラリ構築ツールでオプション-head=fenv を指定する必要があります。

表 2-5 浮動小数点環境ライブラリ

種別	定義名	説明
型(マクロ)	fenv_t	浮動小数点環境全体の型です。
	fexcept_t	浮動小数点状態フラグの型です。
定数(マクロ)	FE_DIVBYZERO	浮動小数点例外をサポートするときに定義されるマクロです。
	FE_INEXACT	
	FE_INVALID	
	FE_OVERFLOW	
	FE_UNDERFLOW	
	FE_ALL_EXCEPT	
定数(マクロ)	FE_DOWNWARD	浮動小数点数の丸め方向のマクロです。
	FE_TONEAREST	
	FE_TOWARDZERO	
	FE_UPWARD	
定数(マクロ)	FE_DEF_ENV	プログラム既定の浮動小数点環境です。
関数	feclearexcept	浮動小数点例外のクリアを試みます。
	fegetexceptflag	浮動小数点フラグの状態のオブジェクトへの格納を試みます。
	feraiseexcept	浮動小数点例外の生成を試みます。
	fesetexceptflag	浮動小数点フラグのセットを試みます。
	fetestexcept	浮動小数点フラグがセットされているか確認します。
	fegetround	丸め方向を取得します。
	fesetround	丸め方向を設定します。
	fegetenv	浮動小数点環境の取得を試みます。
	feholdexcept	浮動小数点環境を保存し、浮動小数点状態フラグをクリアし、浮動小数点例外について無停止モードに設定します。
	fesetenv	浮動小数点環境の設定を試みます。
	feupdateenv	浮動小数点例外の自動記憶域への保存、浮動小数点環境の設定、保存していた浮動小数点例外の生成を試みます。

■使用例

浮動小数点例外のクリアを試みる例です。

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
int ret, e;

void func()
{
    ret = feclearexcept(e);
}
```

2.5.3 inttypes.h

■ポイント

整数型の書式変換を行うライブラリが追加されました。

■説明

inttypes.h は、整数型を拡張するための標準インクルードファイルです。また、stdint.h で定義された整数型を扱うための書式を提供します。以下にマクロと関数の一覧を示します。

書式変換ライブラリを使用するには、標準ライブラリ構築ツールでオプション-head=inttypes を指定する必要があります。

表 2-6 整数型の書式変換ライブラリ

種別	定義名	説明
型(マクロ)	imaxdiv_t	imaxdiv **関数の返す値の型です。**
変数(マクロ)	PRId*N*	printf **関数の書式です。**
	PRIdLEAST*N*	
	PRIdFAST*N*	
	PRIdMAX	
	PRIdPTR	
	PRIi*N*	
	PRIiLEAST*N*	
	PRIiFAST*N*	
	PRIiMAX	
	PRIiPTR	
	PRIo*N*	
	PRIoLEAST*N*	
	PRIoFAST*N*	
	PRIoMAX	
	PRIoPTR	
	PRIu*N*	
	PRIuLEAST*N*	
	PRIuFAST*N*	
	PRIuMAX	
	PRIuPTR	
	PRIx*N*	
	PRIxLEAST*N*	
	PRIxFAST*N*	
	PRIxMAX	
	PRIxPTR	
	PRIX*N*	
	PRIXLEAST*N*	
	PRIXFAST*N*	
	PRIXMAX	
	PRIXPTR	
	SCNd*N*	
	SCNdLEAST*N*	
	SCNdFAST*N*	
	SCNdMAX	
	SCNdPTR	
	SCNi*N*	
	SCNiLEAST*N*	
	SCNiFAST*N*	

	SCNiMAX	
	SCNiPTR	
	SCNo*N*	
	SCNoLEAST*N*	
	SCNoFAST*N*	
	SCNoMAX	
	SCNoPTR	
	SCNu*N*	
	SCNuLEAST*N*	
	SCNuFAST*N*	
	SCNuMAX	
	SCNuPTR	
	SCNx*N*	
	SCNxLEAST*N*	
	SCNxFAST*N*	
	SCNxMAX	
	SCNxPTR	
関数	imaxabs	**絶対値を計算します。**
	imaxdiv	**商、剰余を計算します。**
	strtoimax	**文字列の最初の部分を** intmax_t **型および** uintmax_t **型表現に変換する以外は、**strtol**、**strtoll**、**strtoul **および** strtoull **関数と等価です。**
	strtoumax	
	wcstoimax	**ワイド文字列の最初の部分を** intmax_t **型および** uintmax_t **型表現に変換する以外は、**wcstol**、**wcstoll**、**wcstoul **および** wcstoull **関数と等価です。**
	wcstoumax	

注：*N* **は、**8**、**16**、**32 **のいずれかです。マクロによって** 64 **もあります。**

■使用例

int_latest32_t 型の変数 a を出力する例です。printf の結果は、”value : 30”になります。

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

void func()
{
    int_least32_t a;
    a = 30;
    printf("value : %"PRIdLEAST32"¥n", a);
}
```

2.5.4 stdbool.h

■ポイント

論理型、および論理値に関するマクロを定義する標準インクルードファイルが追加されました。

■説明

bool, true, false の名前を使用することができます。stdbool.h はマクロ名の定義だけからなるインクルードファイルです。以下に定義の一覧を示します。

表 2-7 論理型マクロ

種別	定義名	説明
マクロ(変数)	bool	_Bool **に展開します。**
マクロ(定数)	true	1 **に展開します。**
	false	0 **に展開します。**
	__bool_true_false_are_defined	1 **に展開します。**

■使用例

stdbool.h の使用例です。

```
#include <stdbool.h>

bool a;
int func()
{
    if (a == true) {
        return 1;
    }
    return 0;
}
```

2.5.5 stdint.h

■ポイント

指定した幅の整数型を宣言するための標準インクルードファイルが追加されました。

■説明

整数型を条件ごとに分類して定義します。整数型の大きさが環境に依存しないため、他の環境への移植性が高くなります。stdint.h はマクロ名の定義だけからなるインクルードファイルです。以下に定義の一覧を示します。

表 2-8 指定幅整数型マクロ

種別	定義名	説明
マクロ	int_least*N*_t	8,16,32 および 64 ビットに対する、それぞれの符号あり/なし整数型を少なくとも格納できる大きさを持つ型です。
	unt_least*N*_t	
	int_fast*N*_t	8,16,32 および 64 ビットに対する、それぞれの符号あり/なし整数型を最速で演算できる型です。
	uint_fast*N*_t	
	intptr_t	void へのポインタを相互変換可能な符号あり/なし整数型です。
	uintptr_t	
	intmax_t	すべての符号あり/なし整数型のすべての値を表現可能な符号あり/なし整数型です。
	uintmax_t	
	int*N*_t	N ビットの幅をもつ符号あり/なし整数型です。
	uint*N*_t	
	INT*N*_MIN	幅指定符号あり整数型の最小値です。
	INT*N*_MAX	幅指定符号あり整数型の最大値です。
	UINT*N*_MAX	幅指定符号なし整数型の最大値です。
	INT_LEAST*N*_MIN	最小幅指定符号あり整数型の最小値です。
	INT_LEAST*N*_MAX	最小幅指定符号あり整数型の最大値です。
	UINT_LEAST*N*_MAX	最小幅指定符号なし整数型の最大値です。
	INT_FAST*N*_MIN	最速最小幅指定符号あり整数型の最小値です。
	INT_FAST*N*_MAX	最速最小幅指定符号あり整数型の最大値です。
	UINT_FAST*N*_MAX	最速最小幅指定符号なし整数型の最大値です。
	INTPTR_MIN	ポインタ保持可能な符号あり整数型の最小値です。
	INTPTR_MAX	ポインタ保持可能な符号あり整数型の最大値です。
	UINTPTR_MAX	ポインタ保持可能な符号なし整数型の最大値です。
	INTMAX_MIN	最大幅符号あり整数型の最小値です。
	INTMAX_MAX	最大幅符号あり整数型の最大値です。
	UINTMAX_MAX	最大幅符号なし整数型の最大値です。
	PTRDIFF_MIN	-65535
	PTRDIFF_MAX	+65535
	SIG_ATOMIC_MIN	-127
	SIG_ATOMIC_MAX	+127
	SIZE_MAX	65535
	WCHAR_MIN	0
	WCHAR_MAX	65535U
	WINT_MIN	0
	WINT_MAX	4294967295U
関数(マクロ)	INT*N*_C	Int_least*N*_t に対応する整数定数式に展開します。
	UINT*N*_C	uInt_least*N*_t に対応する整数定数式に展開します。
	INT_MAX_C	intmax_t の整数定数式に展開します。
	UINT_MAX_C	uintmax_t の整数定数式に展開します。

注：*N* は、8、16、32 のいずれかです。マクロによって 64 もあります。

■使用例

変数 a は、32 ビットの符号付き整数を少なくとも格納できる型になります。RX ファミリ用 C/C++コンパイラの場合、long 型になります。

```
#include <stdint.h>

int_least32_t a;
```

2.5.6 tgmath.h

■ポイント

型総称マクロを定義する標準インクルードファイルが追加されました。

■説明

tgmath.h をインクルードし、以下の一覧にある数学関数（型総称マクロ）を使用すると、引数の型に対応した関数名に自動的に展開されます。例えば、float 型の引数を sin 関数に渡すと sinf 関数に、complex 型の引数を渡すと csin 関数に展開されます。tgmath.h はマクロ名の定義だけからなるインクルードファイルです。以下にそのマクロ定義を示します。

表 2-9 型総称マクロ

型総称マクロ	math.h の関数	complex.h の関数
acos	acos	cacos
asin	asin	casin
atan	atan	catan
acosh	acosh	cacosh
asinh	asinh	casinh
atanh	atanh	catanh
cos	cos	ccos
sin	sin	csin
tan	tan	ctan
cosh	cosh	ccosh
sinh	sinh	csinh
tanh	tanh	ctanh
exp	exp	cexp
log	log	clog
pow	pow	cpow
sqrt	sqrt	csqrt
fabs	fabs	cfabs
atan2	atan2	-
cbrt	cbrt	-
ceil	ceil	-
copysign	copysign	-
erf	erf	-
erfc	erfc	-
exp2	exp2	-
expm1	expm1	-
fdim	fdim	-
floor	floor	-
fma	fma	-
fmax	fmax	-
fmin	fmin	-
fmod	fmod	-
frexp	frexp	-
hypot	hypot	-
ilogb	ilogb	-
ldexp	ldexp	-
lgamma	lgamma	-
llrint	llrint	-
llround	llround	-
log10	log10	-

log1p	log1p	-
log2	log2	-
logb	logb	-
lrint	lrint	-
lround	lround	-
nearbyint	nearbyint	-
nextafter	nextafter	-
nexttoward	nexttoward	-
remainder	remainder	-
remquo	remquo	-
rint	rint	-
round	round	-
scalbn	scalbn	
scalbln	scalbln	-
tgamma	tgamma	-
trunc	trunc	-
carg	-	carg
cimag	-	cimag
conj	-	conj
cproj	-	cproj
creal	-	creal

■使用例

sin は自動的に sinf に展開されます。

```
#include <tgmath.h>

float f,ret;
void func(){
    ret = sin(f);
}
```

2.6　マクロ

C99 で新たに追加されたマクロについて説明します。

2.6.1　プリデファインドマクロ

■ポイント

__STDC_ISO_10646__、__STDC_IEC_559__、__STDC_IEC_559_COMPLEX__の３つのマクロが追加されました。

■説明

__STDC_ISO_10646__は、wchar_t の表現する文字コードが ISO/IEC 10646 に準拠している場合に定義されます。またマクロの値は、いつ規定された ISO/IEC 10646 に準拠しているか、yyyymmL（yyyy は年、mm は月）の形式で定義されます。RX ファミリ用 C/C++コンパイラでは、C99 の場合は 199712L と定義されます。しかし、C89 では定義されません。

__STDC_IEC_559__は、annex F (IEC60559 浮動小数点)に準拠することを意味します。RX ファミリ用 C/C++コンパイラでは、C99 では定義されますが、C89 では定義されません。

__STDC_IEC_559_COMPLEX__は、annex G (IEC60559 互換複素数)に準拠することを意味します。RX ファミリ用 C/C++コンパイラでは、C99 では定義されますが、C89 では定義されません。

■使用例

(1) __STDC_ISO_10646__

RX ファミリ用 C/C++コンパイラでは、C99 では 199712 が変数 x に格納されますが、C89 では 0 が格納されます。

```
#ifdef __STDC_ISO_10646__
unsigned long x = __STDC_ISO_10646__;
#else
unsigned long x = 0;
#endif
```

(2) __STDC_IEC_559__

RX ファミリ用 C/C++コンパイラでは、C99 ではコンパイルできますが、C89 ではコンパイルエラーになります。

```
#ifdef __STDC_IEC_559__

#else
#error incompatible float
#endif
```

(3) __STDC_IEC_559_COMPLEX__

RX ファミリ用 C/C++コンパイラでは、C99 ではコンパイルできますが、C89 ではコンパイルエラーになります。

```
#ifdef __STDC_IEC_559_COMPLEX__

#else
#error incompatible complex
#endif
```

2.6.2 __func__

■ポイント

記述された位置の関数名に置換されるマクロです。

■説明

__func__は、記述された位置の関数名に置換されます。マクロでも使用できますが、そのマクロを使用している関数名に置換されます。

■使用例

関数 func_001 の__func__は”func_001”に置換され、関数 func_002 の__func__は”func_002”に置換されます。

```
#include<stdio.h>

void func_001()
{
    /* "function : func_001"と出力されます。 */
    printf("function : %s¥n",__func__);
}

void func_002()
{
    /* "function : func_002"と出力されます。 */
    printf("function : %s¥n",__func__);
}
```

マクロで__func__を使用した例です。上記の例と同じ結果になります。

```
#include<stdio.h>

#define DEBUG()  printf("function : %s¥n", __func__)

void func_001()
{
    /* "function : func_001"と出力されます。 */
    DEBUG();
}

void func_002()
{
    /* "function : func_002"と出力されます。 */
    DEBUG();
}
```

2.7 プラグマ

C99 で新たに追加されたプラグマについて説明します。

2.7.1 #pragma STDC FP_CONTRACT

■ポイント

#pragma STDC FP_CONTRACT を使用することにより、浮動小数点演算を省略するかどうか制御できます。

■説明

浮動小数点演算の省略とは、浮動小数点定数値の丸めによる誤差を考慮しない、あるいは、演算結果の精度落ちがあった場合に例外が発生してもそれを通知しない、といった処理の省略を意味します。「#pragma STDC FP_CONTRACT ON」が宣言されると、それ以降の浮動小数点演算の省略が許されます。一方、「#pragma STDC FP_CONTRACT OFF」が宣言されると、それ以降の浮動小数点演算の省略が禁止されます。「#pragma STDC FP_CONTRACT DEFAULT」が宣言されると、それ以降の浮動小数点演算の省略はデフォルトの指定に戻ります。デフォルトの指定が ON か OFF のどちらに従うかは処理系定義です。

RX ファミリ用 C/C++コンパイラでは、#pragma STDC FP_CONTRACT 指定があっても、無視されます。

■使用例

#pragma STDC FP_CONTRACT を使用して浮動小数点演算の省略可否を制御します。

```
float x,y,z;

#pragma STDC FP_CONTRACT ON
/* 浮動小数点演算の省略を許可 */

void func1()
{
    x = y / z;
}

#pragma STDC FP_CONTRACT OFF
/* 浮動小数点演算の省略を禁止 */

void func2()
{
    x = y / z;
}

#pragma STDC FP_CONTRACT DEFAULT
/* 浮動小数点演算の省略はデフォルトに従う */

void func3()
{
    x = y / z;
}
```

2.7.2 #pragma STDC FENV_ACCESS

■ポイント

#pragma STDC FENV_ACCESS を使用することにより、浮動小数点環境へのアクセスを制御できます。

■説明

浮動小数点環境とは、標準インクルードファイル fenv.h で定義される浮動小数点状態フラグや浮動小数点例外フラグなどのことを言います。浮動小数点環境へのアクセスの有無を明示的にコンパイラに知らせることにより、コンパイラが浮動小数点環境に対する最適化を実施しやすくなる場合があります。「#pragma STDC FENV_ACCESS ON」が宣言されると、それ以降、浮動小数点環境にアクセスする可能性があることをコンパイラに知らせます。一方、「#pragma STDC FENV_ACCESS OFF」が宣言されると、それ以降、浮動小数点環境にアクセスしないことをコンパイラに知らせます。「#pragma STDC FENV_ACCESS DEFAULT」が宣言されると、それ以降の浮動小数点環境へのアクセスはデフォルトの指定に戻ります。デフォルトの指定が ON か OFF のどちらに従うかかは処理系定義です。

RX ファミリ用 C/C++コンパイラでは、常に浮動小数点環境へのアクセスがあると解釈し、#pragma STDC FENV_ACCESS の指定は無視されます。

■使用例

#pragma STDC FENV_ACCESS を使用して、浮動小数点環境へのアクセスの有無を制御します。

```
#include <fenv.h>

long a, b;

#pragma STDC FENV_ACCESS ON
/* 浮動小数点環境へのアクセスあり */

void func1()
{
    a = feraiseexcept(b);
}

#pragma STDC FENV_ACCESS OFF
/* 浮動小数点環境へのアクセスなし */

void func2()
{
    a = feraiseexcept(b);
}

#pragma STDC FENV_ACCESS DEFAULT
/* 浮動小数点環境へのアクセスはデフォルトに従う */

void func3()
{
    a = feraiseexcept(b);
}
```

2.7.3 #pragma STDC CX_LIMITED_RANGE

■ポイント

#pragma STDC CX_LIMITED_RANGE を使用することにより、複素数演算に対する数学公式の使用を制御できます。

■説明

複素数の乗算、除算、絶対値に対する数学公式は、無限大の扱いや、適切でないオーバフローやアンダーフローのために、問題があります。#pragma STDC CX_LIMITED_RANGE を使用することにより、数学公式が適用可能であることをコンパイラに知らせます。「#pragma STDC CX_LIMITED_RANGE ON」が宣言されると、それ以降、数学公式を適用可能になります。一方、「#pragma STDC CX_LIMITED_RANGE OFF」が宣言されると、それ以降、数学公式を適用不可能になります。「#pragma STDC CX_LIMITED_RANGE DEFAULT」が宣言されると、それ以降、数学公式の適用可否はデフォルトの指定に戻ります。デフォルトの指定は OFF です。

RX ファミリ用 C/C++コンパイラでは、#pragma STDC CX_LIMITED_RANGE 指定があっても、常に数学公式を使用します。

■使用例

#pragma STDC CX_LIMITED_RANGE を使用して、数学公式の適用可否を制御します。

```
#include <complex.h>

float complex cf1, cf2, cf3;

#pragma STDC CX_LIMITED_RANGE ON
/*数学公式を適用可能*/

void func1()
{
    cf1 = cf2 * cf3;
}

#pragma STDC CX_LIMITED_RANGE OFF
/*数学公式を適用不可能*/

void func2()
{
    cf1 = cf2 * cf3;
}

#pragma STDC CX_LIMITED_RANGE DEFAULT
/*数学公式の適用可否はデフォルトに従う */

void func3()
{
    cf1 = cf2 * cf3;
}
```

3. C89 から C99 へ移行時の注意点

3.1 暗黙の型宣言

■ポイント

暗黙の関数宣言と暗黙の型宣言に対する規定が変更されています。

■説明

C89 では、関数の原型宣言がなくてもその関数を呼び出すことができました。しかし C99 では、規格上、その関数の呼び出しは未定義動作として扱われるようになりました。

また、C89 では、暗黙の型宣言は int 型で扱われました。しかし C99 では、規格上、暗黙の変数宣言は許されなくなりました。

■使用例

(1) 暗黙の関数呼び出し

下記の例では、関数 func において、原型宣言されていない関数 printf を呼び出しています（本来 stdio.h のインクルードが必要です）。RX ファミリ用 C/C++コンパイラで C89 向けにコンパイルした場合、-message オプションを指定すると、C5223（インフォメーションレベル）が出力されます。一方、C99 向けにコンパイルした場合、C5223 は警告レベルで出力されます。

```
void func(void)
{
   printf("message");
}
```

(2) 暗黙の型宣言

下記の例において、C89 では、関数 func の戻り値と仮引数 a の型は、それぞれ int 型とみなされます。RX ファミリ用 C/C++コンパイラで C89 向けにコンパイルした場合、-message オプションを指定すると、戻り値の型が宣言されていないことに対して C5260（インフォメーションメッセージ）が出力されます。一方、C99 でコンパイルした場合、戻り値の型が宣言されていないことに対して C5260（インフォメーションレベル）が出力され、仮引数 a の型が宣言されていないことに対して C6051（警告レベル）が出力されます。下記の例では、C89 でも C99 でも、int func(int a)と扱われます。

```
func(a)
{
   return a;
}
```

3.2　負の整数除算

■ポイント

負の整数除算に対する振る舞いが変わる可能性があります。

■説明

C89 では、整数除算の除数、被除数のうちどちらか一方が負の数の場合、その結果は実装依存になっていました。一方 C99 では、割り算の結果は全て 0 方向に切り捨てることになりました。この結果、負の整数除算を含むプログラムで、動作が変わる可能性があります。

RX ファミリ用 C/C++コンパイラでは、C89 の場合も 0 方向に切り捨てられます。

■使用例

-1/2 の演算結果は計算上は-0.5 ですが、C89 では、コンパイラによって-1 や 0 になる可能性があります。しかし、C99 では必ず 0 になります。

```
int x = -1/2;
```

ホームページとサポート窓口

- ルネサス エレクトロニクスホームページ
  http://japan.renesas.com/
- お問合せ先
  http://japan.renesas.com/inquiry

改訂記録

Rev.	発行日	改訂内容	
		ページ	ポイント
1.00	2010.04.20	—	初版発行

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA機器、通信機器、計測機器、AV機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制するRoHS指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注2. 本資料において使用されている「当社製品」とは、注1において定義された当社の開発、製造製品をいいます。

ルネサス エレクトロニクス株式会社

■営業お問合せ窓口 http://www.renesas.com

※営業お問合せ窓口の住所・電話番号は変更になることがあります。最新情報につきましては、弊社ホームページをご覧ください。

ルネサス エレクトロニクス販売株式会社　〒100-0004　千代田区大手町2-6-2（日本ビル） (03)5201-5307

■技術的なお問合せおよび資料のご請求は下記へどうぞ。
総合お問合せ窓口：http://japan.renesas.com/inquiry