AKI-H8/3664F Tiny(タイニー)マイコン C コンパイラ

動作環境

OS　Windows95以降、メモリ 16Mバイト以上

Windows内のDOSプロンプト上で動作する。

(MS-DOSのみのマシンでは動作しません)。

ＡＫＩ－Ｈ８／３６６４Ｆ
タイニーマイコン専用
Ｃ コンパイラ

Ｈ８／３６６４用Ｃコンパイラをご購入いただきまして、ありがとうございます。

１、ＣＤ内のファイルの内容はつぎのとおりです。

CH38	EXE	：Ｃコンパイラ本体
C38CGN	EXE	：CH38.EXEが使用するコプログラム
C38ASM	EXE	：CH38.EXEが使用するコプログラム
C38FRNT	EXE	：CH38.EXEが使用するコプログラム
C38MID	EXE	：CH38.EXEが使用するコプログラム
C38PEP	EXE	：CH38.EXEが使用するコプログラム
C38HN	LIB	：ライブラリ3664Hノーマルモード用
CTYPE	H	：ヘッダーファイル
ERRNO	H	：ヘッダーファイル
FLOAT	H	：ヘッダーファイル
LIMITS	H	：ヘッダーファイル
MATH	H	：ヘッダーファイル
SETJMP	H	：ヘッダーファイル
STDARG	H	：ヘッダーファイル
STDDEF	H	：ヘッダーファイル
STDIO	H	：ヘッダーファイル
STDLIB	H	：ヘッダーファイル
ASSERT	H	：ヘッダーファイル
INDIRECT	H	：ヘッダーファイル
NO_FLOAT	H	：ヘッダーファイル
MACHINE	H	：ヘッダーファイル
3664F	H	：ヘッダーファイル(I/Oアクセス)
README.TXT		：このテキストです。
ABOUTC.TXT		：Ｃについての説明や、３６６４Ｆ．Ｈの使い方の説明

3664F.HとABOUTC.TXTは繰り返し参照する事が多いので、プリントアウトする事をおすすめします。

サンプルフォルダには、サンプルソースファイルが入っています。
マニュアルフォルダには、ｃコンパイラのマニュアルが入っています。

２、Ｃコンパイラの動作環境
ＯＳ　　ＷＩＮＤＯＷＳ９５以降
メモリ　１６Ｍ以上
ＷＩＮＤＯＷＳ内のＤＯＳプロンプト上で動作する。
ＭＳ－ＤＯＳのみのマシンでは動作しません。

ソースファイルや、リンカー用のサブファイルを製作、編集するには、パソコンで文字を編集するソフト（例えば　ＭＳ－ＤＯＳのエディター、ＷＩＮＤＯＷＳのメモ帳やワードパットなど）をご使用ください。

３、ＣＤのファイルをハードＤＩＳＫにコピーする。
あらかじめ、ハードＤＩＳＫに¥3664Cというフォルダ（デイレクトリ）を作りその中にＣＤの中身をコピーしてください。
リンカー　LNK.EXEをＡＫＩ－Ｈ８/３６６４キットに付属のＣＤからコピーしてください。

(注意)フォルダ名（デイレクトリ名）は小文字１０文字以内にしてください。
長いとコンパイル時にエラーになります。
また、フォルダは　C:¥3664C　の様にドライブの先頭（ルート）においてください。
ＲＯＭ化の為のリンカー、アセンブラ等はこのＣＤには付属していません。ＡＫＩ－Ｈ８/３６６４キットに付属しています。
それをご使用ください。

４、ＲＯＭライター用のファイルを作る例　CTEST.C
（あらかじめサンプルのCTESTフォルダ内のファイルをCH38.EXEとのあるフォルダC:¥3664Cにコピーしてください。）

リセットベクトル、スタックポインタ、割り込みベクトルはアセンブラで書く必要があります。3664RES.MARがそれです。3664RES.MARからつくった3664RES.OBJをリンカーでＣのオブジェクトファイルと合体させます。
Ｃのmain関数は3664RES.OBJが呼び出します。
3664RES.MAR、3664RES.OBJはこのＤＩＳＫのSAMPLE内に入っています。

(1) ＣコンパイラCH38.EXEを使い、CTEST.CからCTEST.OBJをつくる。
MS-DOSのコマンドラインで次のようにキー入力するとCTEST.OBJが 作られます
```
CH38.EXE -INCLUDE=C:¥3664C -CPU=300HN CTEST.C
```
同時にリストファイルctest.lstが作られます。
CTESTC.BATがこの内容ですのでそれをそのまま実行しても同じです。

(2) リンカーLNK.EXEを使い、CTEST.OBJと3664RES.OBJからCTEST.ABSをつくる。
リンカーのサブコマンドとしてCTEST.SUBを指定する。
```
LNK.EXE -SUB=CTEST.SUB
```
CTESTL.BATがこの内容ですのでそれをそのまま実行しても同じです。

(3) 生成された CTEST.ABS をライターで書き込みます。

5、パッチファイルの説明
パッチファイルはＭＳ－ＤＯＳで使用する物で、キーボードから入力し実行させるソフト（例えばＣコンパイラ CH38.EXE)などを、あらかじめその中に書いておきパッチファイルを実行することでキーボード入力と同じ動作をするものです。

パッチファイルを使用すると、インクルードファイルの指定などを、そのつど入力する必要がなく、便利です。

(I)Ｃコンパイラ用パッチファイル

パッチファイルCTESTC.BATの内容は次のようになっています。

```
CH38.EXE -INCLUDE=C:¥3664C -CPU=300HN CTEST.C
```

このパッチファイルは、CTEST.Cをコンパイルする専用のもので、お客さまが製作されたソースファイル(例えばYUUZA.C)に応用するには、次の様に内容を書き替えたファイル(YUUZAC.BAT)を新に作ってください。

YUUZAC.BATの内容例
```
CH38.EXE -INCLUDE=C:¥3664C -CPU=300HN YUUZA.C
```

(II) リンカー用パッチファイル
リンカーはサブコマンドとしてサブファイルを指定しますので、YUUZA.Cに応用するには、
　リンカー用パッチファイルの内容
　サブファイルの内容
を書き替えたファイルを作ってください。

YUUZAL.BATの内容例
```
 LNK.EXE -SUB=YUUZA.SUB
```

YUUZA.SUBの内容例
```
 OUTPUT YUUZA
 PRINT YUUZA
 INPUT 3664RES,YUUZA
 LIB c38hn
 START P(100)
 EXIT
```


6、リンカー用サブファイルの説明
（ＡＫＩ－Ｈ８／３６６４Ｆタイニーマイコンキット付属のリンカーの説明H8SMANをあわせて、ご覧ください。）

リンカーはサブコマンドとしてサブファイルを使用することで、入力するオブジェクトファイル(.OBJ)の指定や、プログラムの開始アドレスを指定します。

----CTEST.SUBの説明-----

OUTPUT ctest	:CTEST.ABSを出力ファイルとする。
PRINT ctest	:CTEST.MAPを出力する。
INPUT 3664RES,ctest	:3664RES.OBJ、CTEST.OBJを入力ファイルとする。
LIB c38hn	:ライブラリファイルはC38HN（ノーマルモード用）
START P(100)	:この行はメモリの開始アドレスを指定します。
EXIT	:SUBファイルの終了

開始アドレス指定は、領域ごとに指定します。
領域には次の領域があります。
プログラムによっては、上の例のようにプログラム領域だけの場合があります。

P　プログラム領域（main関数などプログラム本体）
C　定数領域（CONST型のデータ）
B　未初期化領域（ＲＡＭ領域）
D　初期化データ領域

START P(100),C(2500),B(0F780) の様に指定します。

START P,C(100),B(0F780)　の様に指定すると定数領域 Cは
プログラム領域 Pに続いて割り当てられます

各領域の大きさはＣコンパイラが出力するリストファイルをご覧ください。

■（株）日立製作所のホームページ内Ｔｉｎｙマイコンホームページに
Ｈ８／３００Ｈ　Ｔｉｎｙシリーズ　アプリケーションノートが掲載されています。Ｃ言語でソフトを記述していますので　ご参考にごらんください。

■本ＣＤ内のＣコンパイラマニュアルはＰＤＦ形式のファイルになっています。
パソコンで、表示、印刷するには、Ａｃｒｏｂａｔ Ｒｅａｄｅｒが必要です
Ａｃｒｏｂａｔ Ｒｅａｄｅｒは　ｗｗｗ．ａｄｏｂｅ．ｃｏ．ｊｐでダウンロードできます。

この文書では、H8-3664F 用 C コンパイラを使ってプログラムを作るために必要な C の基礎知識について説明します。H8 のプログラムを作るために必要な最小限の内容ですから、カーニハンとリッチーの「プログラミング言語C」などと一緒に読んでいただいたほうがよいかも知れません。

１、プログラムのおおまかな構造

C は、関数を並べてプログラムを作ります。そのなかでも、main() は必ず必要な関数です。ここから処理が始まります。
まずは「hello, world」を出力するプログラムを見てみましょう。main という関数が定義されています。

```
#include (stdio.h)
main()
{
    printf("hello, world¥n");
}
```

最初の行 #include (stdio.h) は、ここで stdio.h という名前のファイルを読み込む（インクルードする）ことをあらわします。読み込まれるファイル stdio.h には、入出力関数を使うために必要な情報が書き込まれています。main() ので使われている関数 printf() は文字を出力するための関数です。一般的に、関数は名前のあとに () をつけて表します。このカッコの中には、その関数に渡すパラメータを書いておきます。この例の場合の、関数 printf() に対するパラメータは

```
"hello, world¥n"
```

という文字列です。最後の「¥n(¥n)」は改行をあらわします。
関数 printf() は、すでにだれかが作ってくれてライブラリに登録されていますが、自分で作った関数を使うときも、この printf() を使うのと全く同じように記述します。（ライブラリの関数と、自分で作った関数の区別がつかないということです）
このように C では、プログラムを関数に分けて、その関数を呼ぶことで処理が進んでいきます。

printf() の前の空白や、「{」「}」の位置は自由にきめてかまいません。ただし引用符「"」で囲まれた文字列の中で、行をかえることはできません。

実際にコンパイルしてみると、コンパイラは問題ありませんが、リンカでエラーが出ました。画面に文字を出力したり、ファイルアクセスのための ANSI 標準ライブラリ関数は H8/3664F C コンパイラでは使わないようにしましょう。

２、ループと分岐

条件分岐の if 文や、switch、それにループを構成する do - while などです。

２，１ ループ （for、while、do - while）

for 文の例を、これまた有名な温度換算プログラムを使って示します。


```
/* 温度換算プログラム */
#include (stdio.h)
main()
{
    int fahr;
    for (fahr = 0; fahr (= 300; fahr += 20) {
        printf("%3d %6.1f¥n",
            fahr, (5.0 / 9.0) * (fahr - 32));
    }
}
```

最初の行は注釈行です。コンパイラは /* と */ で、挟まれた部分を無視します。注釈は、複数行にまたがってもかまいません。C++ スタイルの注釈で使われる「//」は、H8/3664F C コンパイラではエラーになります。
main() の中の最初の行 int fahr; は、整数型（int）で、fahr という名前の変数を宣言しています。この例のように、C の変数は使う前に宣言しておく必要があります。変数の型には int のほかにいくつか用意されています。整数のバイト数はコンパイラによって異なる場合があります。ここで示した整数のバイト数は H8/3664F C コンパイラの場合です。Limits.h にそれぞれの型に対する最大・最小値が決められています。

```
char        文字（１バイト）
int         整数（２バイト）
short       整数（２バイト）
long        整数（４バイト）
double      浮動小数点数（？バイト）
```

次の for 文は繰り返しの処理で使います。この例の場合では fahr を０から 300 まで、２０ずつ増やしながらループしています。fahr += 20; は、fahr = fahr + 20; の意味です。

for 文の一般形と、for と同様の繰り返し処理を記述するのに使われる while 文の一般形を示します。この２つは同じ処理になります。まず式１が処理され、式２を処理し、その結果が真（０以外）であれば分を実行し、最後に式３が処理されます。式２は x (y などの関係式で、式１と３は代入文や関数呼び出しになります。

```
(( for 文))
for  (式１; 式２; 式３)
    文

(( while 文))
式１;
while (式２) {
    文
```

```
    式３、
}
```

もう一つの繰り返し処理 do - while は次の形になります。

```
((do - while))
do
    文
while (式);
```

do - while は、文が実行されてから終了条件の式を評価します。前の for と while は、式を評価してから文が実行されましたから、順番が逆です。必要に応じて使い分けましょう。

for の後の文と、do と while の間の文がひとつであれば { } は必要ありません。しかし、文が追加されることがよくあります。このとき、{ } を付けないままにしておくと、思わぬ処理内容になってしまうというバグを発生させますから、文が一つでも { } を付けるようにしておいたほうがよいでしょう。

たとえば、次のプログラムは for ループの中に y += x; と z *= x; の２つの文が入っているように見えますが、y += x; だけが繰り返し処理されて、ループが終了したところで z *= x; が一度だけ処理されます。

```
y = 0;
z = 0;
for (x = 0; x < 10; x++)
    y += x;
    z *= x;         /* ここはループの外 */
printf("%d, %d¥n", y, z);
```

２，２ 分岐 (if、if - else、if - else if、switch)

ある判定をして、その結果で処理を変えるときには if 文を使います。その一般的な形は次のようになります。

```
if (式)
    文１
else
    文２
```

if 文は、式を評価しそれが真（０でない）なら文１を、偽（０）なら分２が実行されます。else 以下は省略可能です。

もしｘが１なら式１、２なら式２、３なら式３，それ以外なら式４、といった場合は else - if を使います。

```
if (x == 1) {
    式１;
} else if (x == 2)
    式２;
} else if (x == 3) {
    式３;
} else {
    式４;
}
```

判定式 x == 1 は、ｘが１と等しければ真（０以外）、等しくなければ偽（０）になります。よくやる間違いで、x == 1 とするところを、x = 1 と書いてしまうことがあります。C では式も値を持ちますから x = 1 自体は値１になり、文法上は問題ありません。しかし、x = 1 は比較ではなく代入文ですから、x の値が１になってしまい、思わぬ結果をもたらします。最近のコンパイラは判定式を書くべきところで代入していると警告を出してくれますが、この H8/3664F C コンパイラは何も文句はいいません。

if と else の間に { } を置いていますが、式が一つだけであれば省力可能です。でも、つけておいたほうが無難です。

同じ変数に対して、いくつかの定数値と比較し分岐する場合は switch 文を使います。switch 文の一般的な形は次のようになります。式の値が定数式と等しいところの文が実行されます。どれにも当てはまらない場合は default の文が実行されます。break や文はなくてもかまいません。 break が無いと、次の case へ処理が進みます。break があると switch を抜けます。

```
switch (式) {
    case 定数式１:
        文１
        break;
    case 定数式２:
        文２;
        break;
    default:
        文３;
        break;
}
```

先ほどの例を if ではなく switch を使って表現してみましょう。
もしｘが１なら式１、２なら式２、３なら式３，それ以外なら式４を実行します。

```
switch (x) {
    case 1:     /* x が１のときの処理 */
```

```
        式１;
        break;
    case 2:     /* x が２のときの処理 */
        式２;
        break;
    case 3:     /* x が３のときの処理 */
        式３;
        break;
    default:    /* x がそれ以外のときの処理 */
        式４;
        break;
}
```

もしｘが１か２なら式１，３なら式３、それ以外なら式４を実行するという場合は、次のようになります。

```
switch (x) {
    case 1:     /* x が１のときの処理 */
    case 2:     /* x が２のときの処理 */
        式１;
        break;
    case 3:     /* x が３のときの処理 */
        式３;
        break;
    default:    /* x がそれ以外のときの処理 */
        式４;
        break;
}
```

３、関数
プログラムをいくつかの関数に分割します。それらの関数が集まって、プログラムになります。最も簡単な関数は次のような形になります。

```
smallFunc() {}
```

関数名 smallFunc の後ろにパラメータを記入するカッコ、そして関数の内容を記述する { } が続きます。この場合は、パラメータが無く、内容も無い関数になっています。

具体的なプログラムを示して説明します。このプログラムはいよいよ H8/3664 で動作するものです。シリアルポート（SCI3）から１文字読み込んでは、シリアルポートから出力するというものです。その動作は、パソコンのターミナルソフトでチェックできます。
main() と inch() だけを残して、後は説明の邪魔になるのでよけました。（後ほど全リストを示します）

include が３つあります。このプログラムをコンパイルするために必要な情報が、ここで読み込まれます。3664f.h は、H8/3664F の I/O ポートなどの定義をしているコンパイラ付属のインクルードファイルです。ただし、ANSI 標準ライブラリではありませんから、C の本を見てもその説明はありません。使い方は難しいのですが、これを使うとスマートに I/O ポート操作を記述することができます。
ctype.h は、よく使う型の定義をしています。これもコンパイラ付属のインクルードファイルです。
myType.h は、このプログラム独自のインクルードファイルで、Sint16 などの、自分で作成した型の定義をしています。

```
#include "3664f.h"
#include "ctype.h"
#include "myType.h"

/* PROTOTYPE */
void initio(void);
Sint16 inch(void);
Sint16 inchxx(void);
void outch(short ch);
void out16h(Uint32 xx);
void out4h(Uint16 xx);
void out2h(Uint8 xx);
void out8h(Uint32 xx);
void outh(Uint8 xx);
void crlf(void);
```

突然出てきた関数プロトタイプは、このファイルの中で使われる関数を前もって宣言しておくものです。

そして main 関数です。みなれない void で囲まれています。main の左側の void は、この関数 main は void 型の値を返すということを意味していますし、main の右側のカッコの中にある void は、void 型のパラメータひとつを必要としているということになるんですね、一応。で、なぜ一応かというと、void というのは特別な型で、void 型を返すということは何も返さないし、void 型のパラメータを必要としているということは、パラメータを必要としない、という、ちょっとひねくれた表現なのでした。

```
void main(void)
{
    Sint16 ch;

    initio();
    crlf();
    outch('*');
```

```
    do {
        ch = inch();
        outch(ch);
    } while (true);
}   /* main */
```

前にもいいましたように、このプログラムはシリアルポート（SCI3）から１文字読み込んでは、シリアルポートから出力するというものです。実際には、読み込んだ文字を大文字に変換して、出力しています。
それでは、main() の中を順番に見ていきましょう。
まず、この関数で使う変数の宣言です。Sint16 型の変数を一つ宣言しています。その変数の名前は ch です。

```
Sint16 ch;
```

文字なのになぜ char 型ではなく、Sint16 という１６ビットの変数になっているのでしょうか。unicode だからというわけではありません。通常、ループの終了をチェックするための文字の終わり（ファイルの終わり）を表す EOF (End Of File) が、char 型では表現できないのです。ここでは EOF チェックをしていませんから、関係はないのですが、入力する文字は、８ビットの char ではなく、１６ビット型の変数を使いましょう。
次に３つの関数が並んでいます。I/O の初期化をして、改行を出力してから、プロンプト（*）を出力します。C の伝統では、関数名は小文字、定数は大文字となっていますが、そんなこと気にせず initio は InitIO、init_IO などとしてもかまいません。大切なのは、スタイルを統一するということです。

```
initio();
crlf();
outch('*');
```

次は do - while ループです。while のカッコの中が true （真）になっています。do - while は、while のカッコの中が真であるあいだはループしつづけますから、これは結局、無限にループをくり返すということです。
ループの中は、一文字入力して、それを出力しています。

```
do {
    ch = inch();
    outch(ch);
} while (true);
```

次にもう一つの関数 inch() を見てみましょう。

```
Sint16 inch(void)
```


{

です。パラメータなしで、Sint16 型の値を返す関数です。Sint16 というのは、符号付き１６ビット整数の意味です。自分で作って myTypes.h で宣言してあります。ほかにもご想像のとおり、Uint16 とか、Sint32、Sint8 などがあります。こうやって、自分につごうのいい型を作ることもできます。

```
Sint16 inch(void)
{
    Sint16 ch;

    ch = inchxx();
    if (islower(ch)) {
        ch = toupper(ch);
    }
    return (ch);
}   /* inch */
```

さて、inch() の内部に目をやると、先ほどの main() と同じ Sint16 型の変数 ch を宣言しています。同じ型同じ名前なのですが、main() の中で使われる変数 ch とは別のものです。{} の中で宣言された変数は、その外の変数とは独立しているのです。

```
Sint16 ch;
```

次の４行は、関数 inchxx() の返した値が小文字なら大文字にする、というところです。islower() と toupper() は、コンパイラ付属のインクルードファイル ctype.h で宣言されている ANSI 標準ライブラリ関数です。これは、書店で販売されている C の教科書に、説明が載っているということを意味します。
inchxx() は自分で作った関数で、islower() と toupper() は、はじめからライブラリに用意されている関数だなんて、ちょっと見ただけでは（よく見ても）分かりません。ある関数があったら、その関数がどこにあるのか探せる環境を作ってください。

islower() と toupper() は、ANSI 標準ライブラリ関数です。この関数のプロトタイプは、このファイルに含まれていません。では、いったいどこにあるのでしょう。はい、それは、最初にインクルードした ctype.h に含まれるのです。
toupper を検索して、それが ctype.h にあるということがすぐにわかるようにしておきましょう。

たとえば、ゲームを作るときに必ず必要となる乱数を発生する関数 rand() をプログラムの中で使ったとします。その rand() のプロトタイプ宣言が含まれるヘッダファイルを、そのプログラムの最初でインクルードしなければなりません。rand() を使うためには、その関数がどのヘッダファイルで宣言されているのか調べなければならないのです。

話は戻りまして、inch() の中の話です。関数 inchxx() の返した値が小文字なら大文字にするというところです。inchxx() は、SCI3 から一文字入力し、その文字を返す関数です。その中身については、最後にソースリストを載せておきましたので、御参照ください。 if

文の判定式部分にある islower(ch) は、渡した値が小文字なら真になりますから、ch が小文字なら if 文の中が実行され、 ch を toupper(ch) の値にします。toupper(ch) は、ch を大文字にした値を返します。

```
ch = inchxx();
if (islower(ch)) {
    ch = toupper(ch);
}
```

最後の return (ch); は、関数のように見えますが、関数ではありません。実はカッコは不要なのですが、なんとなくカッコを付けたくなるんですね。ここは、この関数が ch の値を返すということです。

```
return (ch);
```

ま、このように、プログラムを「I/O を初期化するところ」「一文字入力するところ」「一文字出力するところ」などに分けたものを関数にして、その関数を組み合わせてプログラムを作っていきます。
プログラムをわかりやすくするために一つの関数の行数はすくなめのほうがよいようです。たとえば一つの画面に表示されるくらいとか、プリントして１ページくらいなど・・・

このプログラムの全リストはこの文書の最後に載せておきます。

４、変数の型
４、１　型の種類
いままでいくつかの変数の型が出てきました。そもそも C に用意されている型は次のものがあります。

char	１バイトの文字
int	整数
float	単精度浮動小数点数
double	倍精度浮動小数点数

さらに修飾子として、short と long、さらに signed と unsigned があります。
H8/3664F C コンパイラでは int と short int は１６ビット、long int は３２ビットです。ここで、short int や、long int は通常 int が省略されて short と long となります。short char や、long double がサポートされているか実験してみましょう。

４、２　型変換（キャスト）
変数には、int とか char とかの型があります。ま、アセンブラ的に考えると、型というのは、変数のサイズのようなものです。（同じサイズでも型が違うということもありますから、この説明は正確には間違いです。）
サイズが異なる変数間で代入が行われる場合、型変換（キャスト）が発生します。自動的にコンパイラが型変換してくれるときと、エラーになる場合があります。故意に型変換を行わせたい場合には、キャスト演算子を使います。

型変換が必要なときには変換したい式にキャスト演算子をつけます。キャスト演算子は、変換先の型をカッコで括ったものです。例を次に示します。ここでは、int 型の変数 xx を型変更し、char 型にしています。
構造体を int に変換する、という場合のように、キャスト演算子をつけても変換できず、エラーになる場合もあります。

```
int xx;
char yy;

xx = 100;
yy = (char)xx;
```

４、３　定数
定数は次のように宣言します。この例では、kMaxNumber が 200 に、kTestString は "Test String" に、kLongVal は 40000 になります。
40000L の L は、Long の L で、長整数型の文字列として扱われます。

```
#define kMaxNumber  2000
#define kTestStr    "Test String"
#define kLongVal    40000L
#define kCRcode     0x0d
#define kTABcode    '¥t'
```

アセンブラで使われる１６進数は 0x あるいは 0X を付けてあらわします。たとえば、アセンブラでの１６進数 h'12ab は、0x12ab となります。0x12abL とすると、長整数型（H8/3664 の C コンパイラでは３２ビット） になります。この例では、kCRcode が１６進数です。

文字列と文字は区別されます。２重引用符「"」で囲まれていれば文字列、単一引用符「'」であれば文字です。文字は数値なのですが、文字列は（後で説明しますが）ポインタですから、普通の値のように扱うことはできません。

'¥t'（"¥t"）はタブコードをあらわします。このほかにも改行 ¥n などがあります。

列挙定数というものもあります。定数なんですが、ひとつひとつの値自体は何でもかまわないという時に使います。
たとえば、真と偽を意味する true と false です。

```
enum { false, true };
```

こうすると、false は０に、true は１になります。
次の例のように、値を指定することもできます。shirobon は、kurobon + 1 になります。aobon は 100 です。

```
enum { kurobon = 1, shirobon, akabon, aobon = 100 };
```

定数のように #define を使うマクロがあります。簡単な関数をマクロで表現することができます。マクロを使った関数は、実際に関数ではありません。マクロを使った場所で展開されます。三項演算子の説明を参照してください。

４，４ 演算子

２項演算子には＋、－、＊、／、％があります。％は余りです。 7 % 3 は 1 になります。

比較などに使われる ＞、＜、＞＝、＜＝、＝＝、！＝ もあります。

C++ という具合に、言語の名前にもなってしまったインクリメントとデクリメント演算子は、C++ であれば、 C = C + 1 と同じ意味です。C += 1 と書くこともできます。

```
x = 0;
y = 5;
z = x++ + --y;
```

さて、x、y、z はどんな値になっているでしょうか。注意するところは、インクリメント演算子とデクリメント演算子が、変数のどちらについているかということです。x++ のように、右側についている場合は、x の値を参照してから x を更新します。左側であれば、更新してから、その値を参照します。ですから、z = x++ + --y; は、z = 0 + 4; となって、z は４になります。そして、z に代入後、zx は１に、y は４になります。

アセンブラ的な演算子として、ビット演算子があります。

```
&    ビットごとのＡＮＤ
|    ビットごとのＯＲ
^    ビットごとの Ex-OR
<<   左シフト
>>   右シフト
～   １の補数
```

あまり好きではないのですが、三項演算子 ? : があります。if 文の代わりですが、複数行にわたる if 文が、一つの式で表現できてしまいます。
たとえば、前に出てきた大文字に変換する式は次のようになります。

```
upch = islower(c) ? c - 'a' + 'A' : c;
```

一般的な形は次のようになっています。式１が真であれば式２が、偽であれば式３が全体の式の値になります。

式１ ? 式２ : 式３

先ほどの大文字変換の式を if 文に直すと次のようになります。

```
if (islower(c)) {
    upch = c - 'a' + 'A';
} else {
    upch = c;
}
```

定数のように #define を使うマクロについて説明しておきます。簡単な関数をマクロで表現することができます。マクロを使った関数は、実際に関数ではありません。マクロを使った場所で展開されます。なんだか分からないと思うので、例を示します。

```
#define toupper(c)  (islower(c) ? ((c)-'a'+'A') : (c))
```

先ほどの、大文字に変換する式をマクロ関数にしました。こうすると、toupper() を普通の関数のように使うことができます。ちょっと注意が必要なんですが・・・
この定義を見て、余計な () が増えたことにお気づきでしょう。これは、() が無い場合に、c に相当する部分が、複雑な式だったりすると、マクロを展開したときに意味が変わってしまうことがあるためです。自分でマクロを作るときには注意しましょう。

また、マクロにはもう一つ注意しなければならないことがあります。次の式はどのようになるでしょう。

```
x = 'a';
y = toupper(++x);
```

toupper() が純粋な関数であれば y は 'B' になるはずです。ところが、マクロですから展開されると次のようになります。

```
x = 'a';
y = (islower(++x) ? ((++x) - 'a' + 'A') : (++x));
```

y がどんな値になるか考えてみましょう。++x が一箇所のように見えますが、マクロが展開されると３箇所にもなっています。

５、構造体と共有体 （struct、union）

良く使います。これが分からないと、プログラムが作れません。
H8/3664F の I/O ポートの定義をしている 3664I.h をちょっと見てください。長いので途中まで次に示します。

```
struct st_sci3 {                                    /* struct SCI3  */
              union {                               /* SMR          */
                    unsigned char BYTE;             /*  Byte Access */
                    struct {                        /*  Bit  Access */
```

```
                    unsigned char COM :1;       /*    COM       */
                    unsigned char CHR :1;       /*    CHR       */
                    unsigned char PE  :1;       /*    PE        */
                    unsigned char PM  :1;       /*    PM        */
                    unsigned char STOP:1;       /*    STOP      */
                    unsigned char MP  :1;       /*    MP        */
                    unsigned char CKS :2;       /*    CKS       */
```

このように、struct と union の嵐となっています。

5，1 構造体 （struct）

パラレルポートを考えましょう。ポート１は８ビットのレジスタ２つから構成されます。ひとつはポートコントロールレジスタ（PCR1）で、もうひとつはデータレジスタ（PDR1）です。

それを（3664I.h の定義とは異なりますが）構造体で表現すると次のようになります。
#define のところは、こーゆーもんだと思っていてください。

```
struct st_io {                                      /* struct IO    */
            union {                                 /* PDR1         */
                  unsigned char PDR1;               /*              */
                  unsigned char PCR1;               /* PCR1         */
};                                                  /*              */
#define IO      (*(volatile struct st_io    *)0xFFD0)   /* IO    Address*/
```

define 文と組み合わせることで、ポート１を使うにはこんなふうにします。
PCR1 に 0xff を書き込んで、８ビットすべてを出力ポートにして、PDR1 を０にすることで、ポート１から０を出力します。

```
IO.PCR1 = 0xff;
IO.PDR1 = 0;
```

このように、構造体を使うといくつかの変数をまとめて扱うことができるのです。

構造体の中の変数をメンバーといいます。

よく、例として使われるのが、座標情報です。２次元平面でのｘｙ空間での点を次のように表現できます。

```
struct point {
    int h;
    int v;
};
```

このままなら、こんなふうに使います。point 構造体型の変数 pt を用意し、座標を｛100，250｝にして、線をひく関数 lineto() に渡しています。


```
void lineto(struct point pt);       // プロトタイプ宣言

void lineto(struct point pt)
{
    /* それなりの処理 */
}

main()
{
    struct point pt;

    pt.h = 100;
    pt.v = 250;
    lineto(pt);
}
```

struct といちいち書くのはわずらわしいので、通常は次のように typedef と組み合わせます。

```
typedef struct {
    int h;
    int v;
} point;

void lineto(point pt);  /* prototype */

main()
{
    point pt;

    pt.h = 100;
    pt.v = 250;
    lineto(pt);
}

void lineto(point pt)
{
    /* それなりの処理 */
}
```

構造体は、C とならぶ構造化言語として有名な Pacal ではレコード型といいます。住所録や、会員名簿の個人データなど、いくつかの項目（フィールド）がひと組のデータとして扱われるときに使われるのです。

ひとつの構造体中のデータが、全て同じ型であれば配列でも表現できますが、構造体は、ひとつひとつの項目の型が異なっていてもかまいません。

構造体を使うと便利なことがあります。それは、修正が簡単だということです。たとえば、先ほどの、平面座標を表す Point 構造体です。これは水平座標値 h と、垂直座標値 v という２つの値を持っていました。
さて、これを修正します。座標情報に、色を追加するのです。構造体を使っていれば、構造体の定義に色情報を追加し、色の処理を追加します。関数のパラメータは Point 構造体を渡すということで、修正は必要ありません。
もし、線を引く関数 lineto が構造体ではなく、次のように水平座標と垂直座標をバラバラに渡していたとします。

```
void lineto(int h, int v);
```

これに色を渡すように修正するので、

```
void lineto(int h, int v, int color);
```

と、なります。これだけなら簡単ですが、プログラムの中から lineto を使っているところを全てさがし出して、 color を追加しなければなりません。これはたいへんです。

構造体を使っていれば、color を追加する前とあとは次のようになります。

＜＜追加前＞＞
```
void lineto(Point pt);
```

＜＜追加後＞＞
```
void lineto(Point pt);
```

同じなんです。ただし、構造体の宣言が変わります。
```
typedef struct {
    int h;
    int v;
    int color;
} point;
```

５，２ 共有体（union）
さて、次は共有体です。
共有体は構造体と見かけ上ほとんど変わりありませんが、大きな違いがあります。構造体では、各メンバーに独立した領域が確保されますが、共有体では、すべてのメンバーが領域を共有するのです。

何の目的でこんなものが考えだされたのかというと、そもそもはだいたい同じ項目の並んでいる構造体があって、その異なる部分を共有体にすることで、記録に必要な領域を減らそうとしたのです。

アセンブラに近いプログラムを作る場合にでてくる共有体はちょっと違います。つぎの共有体を見て下さい。

```
union charToShort {
    char ch[2];
    short sh;
};
```

こうしておいて、次のプログラムで何がプリントされるでしょう。

```
main()
{
    union charToShort xx;

    xx.sh = 0x1234;
    printf("%02x %02x¥n", xx.ch[0], xx.ch[1]);
}
```

16 ビット変数の xx.sh に 0x1234 が代入され、次に、xx.ch[0] と xx.ch[1] をプリントしています。H8/3664F では、上位桁がメモリの最初に格納されていますから、xx.ch[0] は xx.sh の上位バイト、すなわち 0x12 が、xx.ch[1] には xx.sh の下位バイト、0x23 が入っているのです。

このように、ある領域を１６ビットデータとしてあつかったり、８ビットデータとしてみることが、共有体を使うことで、可能となるのです。

H8/3664F の C コンパイラでは、H8 の I/O 領域の定義に共有体をたくさん使っています。これは、レジスタが１６ビットアクセスのほうがつごうが良かったり、８ビットや、１ビットごとのアクセスのほうが便利だったりするためです。

3664I.h で定義されているパラレルポートＰ１の構造体を、次に示します。
先ほどの例では２バイト目に PDR1 がありましたが、それが共有体になっています。共有体のメンバーは、１バイト単位の unsigned char BYTE と、各ビットごとの構造体です。
このようにすると、１バイトまとめて入出力するときは BYTE を使い、ビット単位なら BIT 構造体を使えばよいのです。
BIT の中にでてくる :1 は、ビットフィールドということで、１ビット単位でアクセスするメンバーです。

```
struct st_io {                                  /* struct IO    */
};                              /*              */
                途中省略

        union {                                 /* PDR1         */
```

```
        unsigned char BYTE;                     /*  Byte Access */
        struct {                                /*  Bit  Access */
                unsigned char B7:1;             /*    Bit 7     */
                unsigned char B6:1;             /*    Bit 6     */
                unsigned char B5:1;             /*    Bit 5     */
                unsigned char B4:1;             /*    Bit 4     */
                unsigned char   :1;             /*    Bit 3     */
                unsigned char B2:1;             /*    Bit 2     */
                unsigned char B1:1;             /*    Bit 1     */
                unsigned char B0:1;             /*    Bit 0     */
                } BIT;                          /*              */
        } PDR1;                                 /*              */
           途中省略
    unsigned char   PCR1;                       /* PCR1         */
```

こんなふうに使います。まず、コントロールレジスタ (PCR1) に 0xff を書き込んで、ポート１を出力ポートに設定します。次に、データレジスタ (PDR1) の８ビット全体に０を出力し、次に、bit 7 を１にしています。

```
IO.PCR1 = 0xff;                 /* bit 7..0 : Output */
IO.PDR1.BYTE = 0;               /*                   */
IO.PDR1.BIT.B7 = 1;             /*                   */
```

6、配列とポインタ
同じようで違う配列とポインタです。
アセンブラ的な考え方をすると理解しやすいところです。
そもそもポインタは何のために使うのでしょう。いろいろあるのですが、ひとつには、C の関数はパラメータを値でわたすということに関係があります。値で渡すということは、その変数のコピーを渡すことです。コピーを修正してもオリジナルには何の影響もありません。ポインタを渡せば、渡された関数はそのポインタが指している場所を操作することができますから、渡した関数と渡された関数とで変数（作業領域）を共有することができます。渡された関数で、値を変更できるのです。
例えば、ファイルからデータを読み込む関数は、読み込んだときに発生したエラーを返しますが、読み込んだデータはパラメータによって指定された場所（ポインタによって指されたところ）へ格納します。というふうに、ポインタを使うと、関数が複数の値を返すこともできるようになります。

変数 x のポインタを &x で表現します。&x は変数 x の番地です。px をポインタとすると、px = &x と書くことができます。x の番地を px に代入しています。
& の逆が * です。*px でポインタ px が指している場所の内容ということになります。x = *px と書くことができます。ポインタ px が指している場所の内容を x に代入しています。

これだけなら話はそう複雑ではないのですが、変数には型があります。これがちょっと、話をややこしくします。型というのは、例えば、H8/3664F では２バイトの int とか、１バイトの char などです。そのため、ポインタにも型があります。int のポインタとか、char のポインタと呼ばれます。*px が int になったり、char になったりするのです。

char 型の変数はそのサイズが１バイトです。int は２バイトになります。char へのポインタの値を＋１すると、１バイト分アドレスが増えます。int へのポインタを＋１すると、int は２バイトなので、２バイト分アドレスが増えます。ここのところは、またあとで説明します。

6，１ ポインタ
int 型の変数 x にたいして、&x は変数 x のポインタです。 p が int へのポインタとすると *p は、ポインタの指している中身の値になります。すなわち int です。p = &x; あるいは x = *p; という式が成立します。
関数が func(int *z) となっていたとき、z は int のポインタです。& がつくとポインタだってさっきいったばかり・・・。もう一度よく見てください。x が int 型のとき、&x が int のポインタです。x だけなら int です。はい、ひと呼吸。 int のポインタを p とすると *p は int になります。それと同じで、func(int *z) も *z が int である、と読むことができます。*z すなわち、z の指している中身が int なんだから、z は int のポインタということです。

２つの整数を交換する関数を考えます。C のパラメータは値渡しなので、値を交換する関数 swap() を swap(int xx, int yy) のようにすると、交換してくれません。値を交換するためには、値の入っている場所を渡さなければなりません。それが、この swap(int *xx, int *yy) です。int *xx は、xx が int 型のポインタであることを意味します。ポインタというのはアドレスということです。
swap() を呼ぶところで swap(&ii, &jj); としています。&ii は、変数 ii へのポインタです。変数 ii の番地と同じです。

```
#include <stdio.h>
void swap(int *xx, int *yy);

void main()
{
    int ii = 10;
    int jj = 20;

    swap(&ii, &jj);
    printf("%d %d¥n", ii, jj);
}
void swap(int *xx, int *yy)
{
    int temp;
```

```
    temp = *xx;
    *xx = *yy;
    *yy = temp;
}
```

6，2 文字列型変数

あ、C には文字列型という型はありませんでした。通常は文字の配列として文字列を実現します。たとえばこんな感じです。'hello!¥n' という文字列を考えてみましょう。

0	1	2	3	4	5	6	7
h	e	l	l	o	!	¥n	¥o
104	101	106	106	111	33	10	0

下の列の数字は文字に対する ASCII コードを１０進数にしたものです。（改行コードは UNIX の改行である 0x0a になってます）
ch を char 型の配列であると宣言します。宣言で値も書いていますから、その値で配列が初期化されます。

```
char ch[] = 'hello!¥n'
```

配列ですから、ch[0] は 'h'、ch[1] は 'e'、ch[2] は 'l' という具合になっています。最後に文字列の終わりを示す０が自動的に入ります。ch[7] が０になります。
ch[8] はなんでしょう？なんだか分かりませんが、ch[8] を使ってもエラーにはなりません。エラーにならないので、これがシステムダウンの原因の一つになることがあります。

文字列のコピーを考えましょう。
strcpy(char *dst, char *src) は、src を dst へコピーします。

```
#include <stdio.h>

void strcopy(char *dst, char *src);

void main()
{
    char str[20];

    strcopy(str, 'hello!¥n');
    printf('%s', str);
}

void strcopy(char *dst, char *src)
```



```
{
    char ch;

    do {
        ch = *src++;
        *dst++ = ch;
    } while (ch != '¥0');
}
```

関数 strcopy(char *dst, char *src) は、そのパラメータとして char へのポインタを２つ必要とします。その２つは文字列で、src 文字列から dst 文字列へコピーします。文字列の最後は '¥0' で終了しているはずですから、'¥0' をコピーしたらおしまいです。

この行を考えてみましょう。

```
ch = *src++;
```

ch は char 型で、src は char のポインタです。*src は char になるので char 型の変数 ch へ代入できます。（ちなみに、ch = src; はコンパイラに型が一致しないぞ！と文句をいわれてしまいます。）

*src++; の ++ はどんな役目なのでしょう。後ろについていますから、値を参照してから更新されます。下の図のように、src が「o」を指していたとします。ch には 'o' が代入されます。その後、インクリメント演算子 ++ によって、*src ではなく、src が +1 され、「!」を指すようになります。

0	1	2	3	4	5	6	7
h	e	l	l	o	!	¥n	¥o
				src			

という具合に、一文字ずつ src 側から dst 側へコピーされていきます。'¥0' をコピーしたところでループは終了します。

コンパイラが出力するアセンブラのソースを見ると勉強になるのですが、残念ながら H8/3664F 用 C コンパイラはアセンブラのソースを出力してくれません。これは大きな欠点です。そのため、このコンパイラを仕事で使おうとする人は少ないでしょう。

文字列は char の配列であるといいつつ、strcopy() は strcopy(char *dst, char *src) というわけで、ポインタを使ってます。ちょっとだらしないです。配列は配列、ポインタはポインタとして区別するべきでした。strcopy(char dst[], char src[]) が、本来の姿でしょうか。でも普通はポインタを使います。

このように、ポインタと配列はだらしなくしていると同じになってしまいます。

ここで、ちょっと問題です。

```
void main()
{
    char str[20];

    strcopy(str, 'hello!¥n');
    printf('%s', str);
}
```

としていましたが、それを次のように修正します。

```
void main()
{
    char *str;

    strcopy(str, 'hello!¥n');
    printf('%s', str);
}
```

char str[20]; を char *src; に変更しました。配列とポインタは同じですから、これでもエラーは発生せずに無事にコンパイルを終了します。
実行すると、char のポインタ str にコピーしようとします。いえ、コピーしてしまいます。
コピー前に str はどこを指しているのでしょう？そこが問題です。どこを差しているのか分からないのです。どこか分からない場所に文字列データがコピーされてしまいます。コピーされた場所はプログラムのスタックエリアかも知れませんし、作業領域あるいは、I/O ポートのある場所かも知れません。そんな状態ではプログラムが正しく動作するわけがありません。

ここは初期化されていないポインタを渡してはいけないのです。どこか文字列をコピーできるだけの領域を指しているポインタを渡すべきです。このように、どこを指しているのか分からないポインタはすぐに作ることができます。でも、配列は（要素が０でない限り）領域込みで定義されるので、どこを指しているのか分からない、ということは発生しません。
ポインタと配列は全く同じではないのです。

6，3 配列
ポインタと配列は同じものではないということを頭の隅に感じながら、今度はポインタと配列は同じであるという説明をします。正確にはポインタでも配列でも同じことを表現できる、ということですね。

＜＜char の配列＞＞

```
char str[100];
char *pStr;
```

であるとき、str は配列の先頭アドレスです。言葉をかえると str は配列の先頭へのポインタと同じ値です。配列の先頭のアドレスは、その文字どおり C で表現すると &(str[0]) です。&str[0] と同じですが、気持ち悪いので &(str[0]) とします。
こんどは、

```
pStr = str;
```

とすると、pStr は配列の先頭へのポインタになります。

```
pStr = &(str[0]);
```

と同じですね。
さて、準備は整いました。配列とポインタが同じことを表現できることを説明します。

配列の最初の要素 str[0] は *pStr と同じです。pStr は配列の先頭へのポインタですから、その中身は最初の要素、すなわち str[0] となるのです。
次の要素はどうなるでしょう。str[1] です。これは、pStr を使うと *(pStr + 1) となります。このように、str[n] は *(pStr + n) と同じことで、配列とポインタは表現の形式が違うけれど、その実体は同じであることが分かります。
実は、この配列が char 、すなわち１バイトの値を要素に持っているから、ポインタ（すなわちアドレス）に１を加えれば、次のアドレス（すなわち配列の次の要素）になる、というのが何となく理解できると思います。
では、char ではなく、int ではどうでしょう。int は２バイトです。

＜＜int の配列の場合・・・＞＞
char の配列と同様のプログラムにします。

```
int intArray[100];
int *intPtr;
```

intArray は、配列 intArray の先頭へのポインタと同じ値になります。
&(intArray[0]) です。

```
intPtr = intArray;
```

こうすると、intPtr は intArray の先頭へのポインタとなります。
配列の最初の要素、intArray[0] と *intPtr は同じものになることに異義はないでしょう。次に、intArray[1] です。char の配列であれば一つの要素は１バイトですから、次の要素はポインタ（アドレス）に１を加えたものになることは直感的に理解できます。実は、配列の型に関係なく intArray[n] は *(intPtr + n) となるのです。

これは嘘きです。intPtr + n の値は intPtr + n ではなく、intPtr + n * 2 になっているのです。

ポインタに対して加えた値、例えば intPtr + n の n は、「n 番地移動」ではなく、あくまでも「n 番目の要素」となり、番地ではないのです。
こう考えると「ポインタと配列は同じだが、ポインタとアドレスは違うものだ」という結論になってしまいます。

int の配列の要素０から９９までの合計を求めるには、ポインタを使って次のようにすることができます。

```
int getSumOfIntArray(int intArray[])
{
    int *intPtr;
    int sum = 0;
    int xx = 0;

    intPtr = intArray;
    for (xx = 0; xx < 100; xx++) {
        sum += *intPtr++;
    }

    return (sum);
}
```

intPtr++ は intPtr を次の要素を指し示すように値をひとつ増加させるという意味です。

配列とポインタは同じようなものなのですが、次の例はエラーとなります。どこがエラーなんでしょうか。
１００個の int 型の要素を持つ array という名前の配列を作りました。

```
int array[100];
```

array だけなら、&(array[0]) と同じものです。配列 array の最初の要素へのポインタと同じ値です。同じ値だからといって、ポインタのように使うことはできません。

```
sum += *array++;
```

*array はエラーになりませんが、 array++ がいけません。array はどこかのメモリに置かれた領域の番地です。その番地は固定ですから array は、定数として扱われるのです。そのため、array++ は定数を変更するわけですから、エラーとなります。

```
    int array[100];
    int sum = 0;
    int xx = 0;

    for (xx = 0; xx < 100; xx++) {
        sum += *array++;  /* ERROR !! */
    }
```


６，４ I/O アクセス

I/O のレジスタに値を設定したり、読み出したりするときにポインタを使います。I/O アクセスを簡単化するために 36641.h というインクルードファイルが用意されていますが、ここではポインタを使って I/O にアクセスしてみましょう。

H8 の I/O は、メモリと同じアドレス空間に置かれていますから、メモリと同様に読み書きします。そこで問題になるのが、対象となる I/O レジスタのサイズによって扱い方が異なるということです。
レジスタが１バイトなら、１バイトの変数となる char あるいは unsigned char へのポインタを使いますし、レジスタが２バイトなら２バイト変数となる int あるいは unsigned int へのポインタを使います。
例えば、パラレルポート１のデータレジスタ (PDR1) に 0xff を書き込むときには次のようにします。
まず、パラレルポート P1 のデータレジスタのアドレスは 0xffd4 ですから、１バイトの変数へのポインタ p1 を用意し、それにアドレス 0xffd4 を代入しておきます。p1 はポインタですから、*p1 はポインタが指している内容になります。そこで、*p1 = 0xff; とすることで、p1 が指している番地の内容を 0xff に設定したことになります。
注釈になっている行は、36641.h で宣言されている構造体を使った I/O アクセスの表現方法です。どちらも同じことをしています。

```
/* IO.PDR1.BYTE = 0xff; */
char *p1dr = (char *)0xffd4;
*p1dr = 0xff;
```

２バイトレジスタは次のようになります。タイマーWの２バイトレジスタ GRA に値を設定しています。タイマーWの GRA は番地 0xff88 です。そこに２バイトの値 0x9c40 を書き込んでいる例です。
注釈になっている行は、36641.h で宣言されている構造体を使った I/O アクセスの表現方法です。どちらも同じことをしています。

```
/* TW.GRA = 0x9c40; */
int  *gra0  = (int  *)0xff88;
*gra0 = 0x9c40;
```

36641.h を使うと、その I/O レジスタのアドレスを調べなくてもよくて便利なのですが、本当にそのアドレスになっているのかアセンブラ出力がないので、確認する方法がありません。プログラムの動作がおかしいときにはポインタを使って、レジスタのアドレスを直接表現したほうが良いでしょう。

7、そのほか
何か書き落としたことはないでしょうか。

C はトリッキーなプログラムを作ることもできます。また、プログラマーはついつい「自分は、こんなこともできるんだぞー。どうだすごいだろう。」的なプログラムを書いてしまいがちです。そんなプログラマーに C はぴったりなんです。
しかし、それではいけません。そんな状態に陥らないように、分かりやすいプログラムを作ることを心がけましょう。三項演算子よりは if 文を使うようにしたり、if 文の条件式には代入文はかかないとか、注意すべきところはたくさんあります。演算子を評価する優先順位を覚えていないと理解できないような式は、() を使って評価の優先順位を明確にすることも大切でしょう。

C で書かれたプログラムは、CPU が変わってもほとんど修正せずに使うことができます。しかし、レジスタの設定など、CPU に依存した記述をプログラムのあちこちにばらまいてはいけません。CPU などの環境が変わっても、その変化に対するプログラムの修正を少なくするためには、I/O の初期化や、ビットのテストなど、その CPU 特有の処理は、なるべく一箇所にまとめておき、簡単に変更できるようにしておきます。
また、入出力を使わない内部処理だけのプログラムであれば、パソコン上の C でチェックできます。デバック環境の貧弱さを十分カバーしてくれます。

ま、そんなわけで、C を好きになるのはかまいませんが、プログラムを作ることが目的であって、C はそのための道具であることをお忘れなく。

最後にシリアルポートから入出力するプログラムの C ソースを示しておきます。

```
/*
 *      sci.c
 *              for H8/3664
 *              set lab 4
 */

#include "3664f.h"
#include "ctype.h"              /* islower() and toupper() */
#include "myType.h"

/* CONST */
#define kCRcode         (0x0d)
#define kLFcode         (0x0a)

/* TYPE */
/* VAR */
/* PROTOTYPE */
void initio(void);
Sint16 inch(void);
Sint16 inchxx(void);
void outch(short ch);
void out8h(Uint32 xx);
void out6h(Uint32 xx);
void out4h(Uint16 xx);
void out2h(Uint8 xx);
void outh(Uint8 xx);
void crlf(void);


/* PROGRAM */
void main(void)
{
        Sint16 ch;

        initio();
        crlf();
        outch('*');
        do {
                ch = inch();
                outch(ch);
        } while (true);
}       /* main */


void initio(void)
{
        Sint16 xx;
        IO.PMR1.BIT.TXD = 1;
        SCI3.SCR3.BYTE = 0;             /* clear all flags      */
        SCI3.SMR.BYTE = 0;              /* Async, 8bit, NoParity, (Even), stop1, 1
/1 */
        SCI3.BRR = 25;                          /* 19200baud (CPU=16MHz) */
        for (xx = 0; xx < 280; xx++) {
                /* wait 1 bit time (1/9600 sec) */
        }
        SCI3.SCR3.BYTE = 0x30;  /* scr=0011 0000 (TE=1,RE=1) */
        xx = SCI3.SSR.BYTE;             /* Dummy Read */
        SCI3.SSR.BYTE = 0x80;   /* Clear Error Flag (TDRE=1) */
}       /* initio */


Sint16 inch(void)
{
        Sint16 ch;
```

```
        ch = inchxx();
        if (islower(ch)) {
                ch = toupper(ch);
        }
        return (ch);
}       /* inch */


Slnt16 inchxx(void)
{
        Slnt16 ch;

        do {
                ch = SCI3.SSR.BYTE;
        } while ((ch & 0x78) == 0);     /* RDRF=ORER=FER=PER=0 */

        if (SCI3.SSR.BIT.RDRF == 0) {
                ch = -1;                        /* error */
                SCI3.SSR.BYTE = 0x80;
        } else {
                ch = SCI3.RDR;
                SCI3.SSR.BIT.RDRF = 0;
        }
        return (ch);
}       /* inchxx */


void outch(short ch)
{
        Slnt16 xx;

        do {
                xx = SCI3.SSR.BIT.TDRE;
        } while (xx == 0);
        SCI3.TDR = ch;
        SCI3.SSR.BIT.TDRE = 0;
}       /* outch */


void out8h(Ulnt32 xx)
{
        out4h((Ulnt16)(xx >> 16));
        out4h((Ulnt16)(xx & 0xffff));
}       /* out8h */
```



```
void out6h(Ulnt32 xx)
{
        out2h((Ulnt8)((xx >> 16) & 0xff));
        out4h((Ulnt16)(xx & 0xff));
}       /* out6h */


void out4h(Ulnt16 xx)
{
        out2h((Ulnt8)(xx >> 8));
        out2h((Ulnt8)(xx & 0xff));
}       /* out4h */


void out2h(Ulnt8 xx)
{
        outh((Ulnt8)(xx >> 4));
        outh((Ulnt8)(xx & 0x01));
}       /* out2h */


void outh(Ulnt8 xx)
{
        xx &= 0x0f;
        xx |= '0';
        if (xx > '9') {
                xx += 7;
        }
        outch(xx);
}       /* outh */


void crlf(void)
{
        outch(kCRcode);
        outch(kLFcode);
}       /* crlf */

/* end of sci.c */
```