C 言語入門ーいかれたオヤジ版 (中途版)

一応は勉強したけれど.. という人へ

2010 年 6 月 27 日　ビットフィールドの間違い訂正

目次

表目次

第 1 章

はじめに

私は職業プログラマーではありませんが、生意気に C 言語の入門書を書いてみようと思います。C 言語は、私の場合 Turbo-C から始まりました。最初は入門書の記事をいれるだけでなかなか自分のプログラムは書けませんでした。そんな状態が何年かすぎて、一応少し書けるようになりました。でも、プログラムは大嫌いです。そんな奴の書く入門書だから眉にたっぷり唾をつけておいてくださいね。まあ、これを書いているうちに、自分の勉強になりました。曖昧でぼんやりしていたものがハッキリしました。

なお、Vine linux 5.0 の環境で書いています。

また、C はプリプロセッサと分離して使われることがないと思いますのでプリプロセッサと一緒にしてここでは C 言語とします。

1.1 参考文献

参考文献 1：C 言語入門　Les Hancock,Morris Krieger 共著　アスキー出版局監訳 1987 年 7 月 1 日　第 2 版

参考文献 2：Borland C++ バージョン 4.0　ライブラリリファレンス　1994 年 9 月 26 日　著者　Borland　international　翻訳・発行　ボーランド株式会社

第 2 章

C 言語のあらまし

たぶん、C 言語はすでに勉強したことは 1 度や 2 度ではないでしょう。それでは、簡単におさらいをやってみましょう。

2.1 #include

C 言語は、自己増殖型言語というべき代物です。ある関数を作るとそれがライブラリとなってどんどん機能が増えてゆきます。ですから、最初はわずかな機能しかありません。もうなんにもできないといった方がいいかも知れません。

したがって、最小限の定義した定義・関数などを取り込む必要があります。それをインクルードするといいます。だから、最初に書く第 1 行目のプログラム行は

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

などと書きます。

最低限、

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
```

と書いておけばいいでしょう。これらの <> で囲まれたものは、特別なディレクトリーを示すことになります。

一般に/usr/include/がそこになります。

では、何が書いてあるのでしょう？一度見てやって下さい。あまりもすごい定義があるのにびっくりするでしょう。たとえば、limits.h には、数の型毎の最大最小が書かれています。

さて、ユーザー定義のインクルードファイルは xxxx.h のように拡張子は.h で

```
#include "xxxx.h"
```

のように”"で区切ります。これは現在ソースのあるのディレクトリになります。

include は、文字通りまるまるファイルを読み込んで貼り付けます。したがって、実際は非常に大きなファイルをコンパイルしていることになります。

2.2 関数の宣言

関数は、

```
int main (void)
{
変数宣言部
本体
return(関数が返す int の値または変数);
}
```

これは、メイン関数でプログラムの実行時に最初によばれる関数です。最初の int は関数が終了して返す値が int という型であるということを示しています。

つぎに main が関数名です。(void) は main が呼び出されるとき、渡されるべき変数達の型と名前を表しています。ここでは void なので何も渡されない関数であることを示しています。

世間では、void はなにも渡したり返したりしないので、書く必要がないと省略されている場合がありますが、流儀として書くことにしています。

次の例では、

```
int kansu (int a[][10],int x,int y)
{
変数宣言部
本体
return(関数が返す int の値または変数);
}
```

関数名が kansu で引き渡される変数は int a[ ][10],int x,int y ですね。この kansu の返す値は int ですね。

では、何も返さない関数はどうなるでしょう。

```
void kansu_1 (int a[][10],int x,int y)
{
変数宣言部
本体
}
```

このように void と宣言します。

では、何も受けとらず、何も返さない関数は

```
void kansu_2 (void)
{
変数宣言部
本体
}
```

当然、このようになりますね。

じゃ、返す値が 1 つでないときは、「どうするのか」とおっしゃりたい? それには、引数をポインタにする、変数を広域宣言にする、構造体を返すなどといった風にします。

2.2.1 関数プロトタイプ宣言

関数の定義の前に関数の宣言行のみ集めてプロトタイプ宣言をします。下の例ではこれがないと func1 の中で func2 が使われていますが、func1 のコンパイル時点で func2 の引数の数と型が一致しているかわかりません。そこで、あらかじめ文頭で宣言します。

```
int func1 (int a);   /* 関数 func1 のプロトタイプ宣言 */
int func2 (int b);   /* 関数 func2 のプロトタイプ宣言 */

int func1 (int a)
{
  int i;

      文のかたまり;
      i=func2(123);  /* コンパイラは上から見てくるので */
                     /* プロトタイプ宣言がないと */
                     /* func2 の引数の数や型がわからない */
      文のかたまり;
}

int func2 (int b)
{
  int i;

      文のかたまり;
}

int main (void)
{
      文のかたまり;
      i=func1(123);
      文のかたまり;
}
```

2.2.2 return

return は、関数から抜けることを意味し、後ろに関数からのもどり値を書きます。

2.2.3 exit

exit は、プログラムから抜けることを意味し、後ろにもどり値を書きます。

```
int main (void)
{
      文のかたまり;
      if (a==0){exit(1);} /* 強制終了 */
      文のかたまり;
}
```

2.3 型

型とは、あらかじめ先ほどの include で決まられているものや、自分で定義した型が指定できますのでこれこれとすべてあげることはできません。ただ、代表的なものは、表 2.1 に示すものがあります。

表 2.1　型

型宣言	意味
char	8bit の文字型
int	32bit の符号つき整数
unsigned int	32bit の符号なし整数
long int	64bit のの符号つき整数
unsigned long int	64bit のの符号なし整数
float	単精度の浮動小数点の数
double	倍精度の浮動小数点の数
enum	列挙型

たとえば、

```
int main (void)
{
int i=0;
char ch;
enum hantei {OK,NG,UNKNOWN} rank[20];
本体
return(関数が返す int の値または変数);
}
```

という風になります。

```
int i=0;
```

このように、変数 i の宣言で i の初期化も行うことができます。

2.3.1 enum

enum は、以下のような構文です。

```
enum 型の名前 {とり得る値のリスト} ; /* enum の型のみ宣言 */
enum 型の名前 {とり得る値のリスト} 変数名; /* enum の型と変数の宣言*/
```

たとえば、hantei はこのようになります。

```
enum hantei {OK,NG、UNKNOWN} rank[20];
```

あるいは、

```
enum hantei {OK,NG、UNKNOWN};
```

```
enum hantei rank[20];
enum hantei buhin[10];
```

この宣言は、hantei は OK か NG か UNKNOWN の 3 つの値を持つことをしています。したがって、次の hantei 型の変数 rank に違う型の代入、例えば rank[1]=i; としたりするとエラーとなりますので、ミスを減らすことができます。したがって、あらかじめとり得る値が決まっているものは enum 宣言しておくべきです。

また、型が違うもの同士は、代入や演算はできません。

2.3.2 キャスト

一時的に型を変更するときキャストを使います。

```
(一時的に変更する型)ch;/* ch は値が呼び出されて一時的に変更する型に変換されます */
```

たとえば、

```
i=i+ch;
```

この場合、i は int 型で ch は char 型でエラーになります。そういうときはキャストをするとエラーになりません。

```
i=i+(int)ch;
```

ここでは (int)ch とキャストしています。つまり、演算する前に「まず int に変換してね」ということです。

2.3.3 typedef

typedef は型を宣言することができます。たとえば、先の enum ですが、いちいち enum と書かねばなりません。typedef なら、hantei という新しい型を作ることができます。

```
typedef enum {OK,NG、UNKNOWN} hantei;
hantei rank[20];
hantei buhin[10];
```

2.4 記憶クラス

データには型のほかに記憶のクラスがあります。それは、自動変数、レジスタ変数、静的変数、外部変数などがあります。ここでは、レジスタ変数については説明しません。

2.4.1 自動変数

変数は、定義された場所によって有効範囲が決まります。もちろんすべて名前を変えれば、ぶつかることはありませんのでその種のバグはありません。しかし、有効範囲がわかっていれば、同じ名前を使うことができます。すると、修正の範囲が明確になります。自動変数とは、関数の中で定義され、関数が呼び出されたとき作られます。したがって、

関数から抜けると消えてしまいます。

このプログラムは、main で 1 から 10 まで発生し func で合計を求めるものつもりです。

```
int func (int a)
{
   int sum=0;

        sum=sum+a;
        return(sum);
 }

int main (void)
{
   int i;
   int ans;

      for (i=0;i<10;i++){
         ans=func(i);
     }
```

この例では、func は呼び出される毎に sum という整数型の変数を作り 0 にします。そして a をたして、つまり、a+0 を返すことになります。したがって、ans は i の値ですから 10 で終わります。

どこで間違ったかというと、sum が func の呼び出しのたびに作られ、抜ける時には消えてしまうことにあるわけです。

また、i,ans も自動変数ですから main が起動して作られ、終わると消滅します。

2.4.2 静的変数

静的変数とは static と宣言された変数です。コンパイル時に作られます。

```
int func (int a)
{
static int sum=0;

        sum=sum+a;
        return(sum);
 }

int main (void)
{
   int i;
   int ans;

      for (i=0;i<10;i++){
         ans=func(i);
     }
```

この場合、この変数はコンパイルするとき作られ、ここでは

```
static int sum=0;
```

と初期化を指定されていますからコンパイル時に 0 にされます。したがって、sum は合計をし続けます。しかし、この sum は func のなかにあり、main からは見れませんし、書き換えもできません。

2.4.3 外部変数

外部変数は、関数の外で定義されます。

```
int sum=0;

int func (int a)
{
       sum=sum+a;
       return(sum);
 }

int main (void)
{
   int i;
   int ans;

      for (i=0;i<10;i++){
         ans=func(i);
     }
```

この場合、sum は、func、main の外にあるので、消えることはありません。また、func,main の両方から、よむことも書くこともできます。

もし、同じ名前の変数があるとすると、自動変数→静的変数→外部変数の順に優先されますので自動変数が一番順位が高くなります。

2.5 定数宣言

定数は、const で始まります。定数宣言された名前には代入ができません。

```
const float pi=3.1415926;
const char *s[]="hello world!\0";
const char cr=0x0d;
```

したがって、定型の文章などを定数宣言すると、便利です。定数なので書き換えはできません。

```
const char *message[3]={"Good morning\0","Good afternoon\0","Good evning\0"};
```

また、以下の場合、初期値は同じですが、書き換えると書き換わってしまいます。

```
static char *message[3]={"Good morning\0","Good afternoon\0","Good evning\0"};
```

2.6　配列と構造体と共用体

2.6.1　配列

配列とは、番号つきのロッカーだと思うといいでしょう。

```
int i_0;
int i_1;
int i_2;
int i_3;
int i_4;
```

このように、宣言すると 5 つの変数が宣言できますが、3 番目の i というには簡単ではありません。そこで配列になるわけです。

```
int i[5];
```

このように宣言すると i は i[0],i[1],i[2],i[3],i[4] の 5 つの番号つきの変数を宣言できます。

```
j=j+1;
i[j]=0;
```

このように、番号を j という変数で指定することもできます。
次のように多次元の配列も使えます。ここでは seiseki という 5 クラスの 40 人の生徒の国語、算数、理科、社会、英語の 5 教科の点数の配列です。

```
int seiseki[5][40][5];
```

2.6.2　構造体

多次元の配列はなるべく構造体にすべきです。というのは、すべて int 型なので点数と生徒の番号が間違えてもわかりませんし、何番の配列が国語だったか忘れてしまうと厄介です。
アセンブラでは、構造体や共用体が使えないと、悲劇的事態になります。つまり、データの管理をすべて自分でおこない、これは先頭から +1 だ、+6 だとかするわけですから、うまくいくはずがありません。
構造体の宣言は、

```
struct 型の名前 {
型　メンバー名１;
型　メンバー名２;
型　メンバー名３;
型　メンバー名４;
型　　・
型　　・
型　　・
};
```

で行います。あるいは、構造体の宣言と変数名までまとめてする以下の形式があります。

```
struct 型の名前 {
```

```
型 メンバー名１;
型 メンバー名２;
型 メンバー名３;
型 メンバー名４;
型     ・
型     ・
型     ・
}変数名;
```

もちろん、構造体がメンバーであっても構いません。

さて、先ほどの例を取り上げると

```
enum kurasu {A,B,C,D,E};
struct seiseki {
enum kurasu kumi;
int kokugo;
int sansu;
int rika;
int syakai;
int eigo;
};
```

と seiseki 構造体を宣言します。enum は組の仕分けです。こう宣言すると seiseki というデータの塊にそれぞれ名前をつけられます。これを構造体のメンバといいます。

```
struct   seiseki seito[40];
```

という風に seito を確保し、たとえば、seito[12] のデータの設定は、

```
seito[12].kumi=A;
seito[12].kokugo=70;
seito[12].sansu=65;
seito[12].rika=85;
seito[12].syakai=65;
seito[12].eigo=55;
```

のように、使います。こうすると間違いにくくなります。構造体のメンバは、ピリオードで区切られて書きます。このピリオードは日本語の「の」だと思えばよいでしょう。seito[12].kokugo は seito[12]「の」kokugo とよめば解りやすいですよね。

構造体のメンバがまた別の構造体でそのメンバはどうなるかというと、

```
日本国.東京都.千代田区.1丁目.1番地.鈴木.一郎.年齢=56;
日本国.東京都.千代田区.1丁目.1番地.鈴木.一郎.性別=man;
```

この例は鈴木一郎さんという人が仮にいたとして、構造体、日本国「の」から始まる多数の構造体のメンバを指定して年齢というメンバに 56 を代入しています。

もちろん、構造体の名前には日本語は使えませんが構造体という仕組みとメンバがわかったと思います。

構造体は配列と違って、要素毎にコピーする必要がありません。構造体としてコピーされます。つまり、構造体のメンバーがまるごとコピーされるのです。配列より便利ですね。

構造体というデータ構造は、明確で曖昧さがなく、行列などの特別な場合を除いて多次

元の配列を置き換えるべきです。

2.6.3 ビットフィールド指定

また、構造体では:を使ってビットフィールドのサイズを指定できます。

```
struct reg{
       unsigned PS:5;
       unsigned PZ:3;
 } ;
struct reg D;
char ch=0x56;
    D.PS= ch & 0x3f;
    D.PZ= (ch >> 5) & 0x7;
```

この例では、5 ビットの PS というものと 3 ビットの PZ を割り当てています。

2.6.4 共用体

構造体と共用体は同じようなものですが、共用体のメンバーは、すべてが重なっています。

```
union aa {
     char c;
     char d;
}a1;
```

この場合、a1.c,a1.d は、同じメモリを使いますので、みな同じ値になります。つまり、同じメモリを違う名前でよんでいるだけです。

```
union ab {
     int  i;
     int  k;
}b1;
```

この場合も、b1.i,b1.k は同じメモリを使いますので、みな同じ値になります。

では、違う型が宣言されるとどうなるでしょう。

```
union rei {
     char c;
     char d;
     int  k;
     int  i;
     double x;
     double y;
     char s[30];
     char t[15];
}a;
```

この例では、a.c,a.d,a.k,a.i,a.x,a.y,a.s,a.t はみな同じメモリを使います。

つまり、わかりやすい例としては、a.c,a.d と a.k,a.i と a.x,a.y と a.s,a.t は名前が違うけどメモリとしてはまったく同じになりますね。a.c と a.k と a.x と a.s ももちろん同じです。

union ではそれぞれのメンバがアクセスできるようにするため、結局 union の領域は一番大きいメンバの型のサイズになります。したがって、union rei a のサイズは一番サイズの大きい s の 30 バイトとなります。

2.6.5 構造体と共用体

たとえば、マイコンのレジスタなどはビット単位に意味があったりしますが、ビット単位でアクセスしたいときがあるし、まとめてバイト単位でアクセスしたいときがあります。こういうとき、共用体が便利です。

これは、Z80 というマイコンの PSW レジスタを定義したものです。BYTE と構造体 BIT は同じメモリを使います。

```
union FLAG{                 /*共用体の名前は FLAG  */
     unsigned char BYTE;     /* バイト単位でアクセスするとき */
     struct{                 /* ビット単位でアクセスするためにビット
フィールドの構造体*/
           unsigned C:1; /* C ビットは 1 ビットのサイズ*/
           unsigned N:1; /* N ビットは 1 ビットのサイズ*/
           unsigned PV:1; /*  PV ビットは 1 ビットのサイズ*/
           unsigned bit3:1;/*  ビットは 1 ビットのサイズ*/
           unsigned H:1;   /*  H ビットは 1 ビットのサイズ*/
           unsigned bit5:1;/*  ビットは 1 ビットのサイズ*/
           unsigned Z:1;   /*  Z ビットは 1 ビットのサイズ*/
           unsigned S:1;   /* MSB S ビットは 1 ビットのサイズ*/
           } BIT;               /*構造体の名前は BIT  */
      };
struct AF {
      unsigned char A;
      union FLAG    F;
      };

struct AF PSW;
```

と宣言しておくと

```
      PSW.A=33;
      Z_FLAG=PSW.F.BIT.Z;
      FLAG=PSW.F.BYTE;
```

というようにビット単位でもバイト単位でもアクセスできます。

2.7 制御構造

制御とは、条件判断でプログラムの流れを変えることです。

1. if
2. for
3. while
4. switch case
5. do while

6. goto
7. break

が、挙げられます。

2.7.1　if

if 文には、2 つの形式があります。はじめの形式は

```
if（条件式）{
 条件式が真のとき実行する文のあつまり;
}
```

次の形式は

```
if（条件式）{
 条件式が真のとき実行する文のあつまり;
}else{
 条件式が偽のとき実行する文のあつまり;
}
```

です。両方ともに実行する文のなかに、文がいくつあっても、また、if 文があっても構いません。

2.7.2　for

for は繰り返しの反復実行を行います。

```
    for(初めに実行される式; 繰り返し判定条件式; 繰り返しの際に実行される
式){
     文のあつまり;
    }
```

という構成をとります。if 文で書くと

```
    初めに実行される式;
label:
     文のあつまり;
    if(繰り返し判定条件式){
    繰り返しの際に実行される式;
    goto   label
    }
```

となります。よく見かけるのは

```
sum=0;
for(i=0;i<10;i++){
 sum=sum+i;
}
```

などですが、

```
for(;;){
文のあつまり;
}
```

これなんかだと、繰り返し判定条件式がありませんので無限ループになります。

```
for(sum=0;(k%356)!=0;j=j*i+3){
 文の集まり;
}
```

これのように初めに実行される式、繰り返し判定条件式、繰り返しの際に実行される式の変数が皆違いますが、問題ありません。

2.7.3 while

while は繰り返しです。

```
while(条件式){
 条件式が真のあいだ繰り返し実行する文のあつまり;
}
```

2.7.4 switch..case

switch..case には 2 つの形式があります。いずれの場合も default のケースは省略することができます。

まずはじめは

```
switch(整数式)
{
   case(整数値 1):
   {
       整数式が整数値 1 と等しいとき実行すべき文の集まり;
   }
   case(整数値 2):
   {
       整数式が整数値 2 と等しいとき実行すべき文の集まり;
   }
   •
   •
   •
   case(整数値 n):
   {
       整数式が整数値 n と等しいとき実行すべき文の集まり;
   }
   default:
   {
       整数式が整数値 1 から整数値 n とどれとも等しくないとき実行すべき
文の集まり;
   }
}
```

整数式は整数を返す関数でも構いません。この場合、整数式がたとえば整数値 2 と等しければ case(整数値 2) 以下のすべての後続する文を実行します。

整数式が整数値 2 と等しいときだけ実行すべき文の集まりのみを実行させるのは次の形式になります。

```
switch(整数式)
{
  case(整数値 1):
  {
      整数式が整数値 1 と等しいとき実行すべき文の集まり;
      break;
  }
  case(整数値 2):
  {
      整数式が整数値 2 と等しいとき実行すべき文の集まり;
      break;
  }
  •
  •
  •
  case(整数値 n):
  {
      整数式が整数値 n と等しいとき実行すべき文の集まり;
      break;
  }
  default:
  {
      整数式が整数値 1 から整数値 n とどれとも等しくないとき実行すべき
文の集まり;
      break;
  }
}
```

実行すべき文の集まりの最後に break; を書かねばなりません。

2.7.5　do..while

do..while は繰り返しです。

```
do
{
 まず一回実行され条件式が真のあいだ繰り返し実行する文のあつまり;
}
while(条件式);
```

while は、条件が満足すると実行しますが、つまり、条件が満足されなければ 1 回も実行されませんが、do..while はまず 1 回実行してから条件がまだ満足していれば繰り返します。そこが違います。

2.7.6　break

ループで break が実行されるとループから抜けます。ループの次の文に移動します。

2.8　演算子

演算子とは端的にいうと＋,-などの演算記号のことです。表 2.2 に示します。結合規則が左から右ということは $5+3+2=8+2=10$ という風に左から評価されるということ

です。もちろん、優先順位は保存されています。

```
if(4==(c=b=a=5)){
```

とかくと、a=5 が行われ, つぎに b=a が行われ, つぎに c=b が行われ, そののち if(4==c) となります。すると右から左のように感じますが、これは代入であって、ここでの演算子は == なのです。

```
a=5;
b=a;
c=b;
if(4==c){
```

ということです。

2.8.1 sizeof 演算子

sizeof 演算子は、変数または型のメモリサイズつまりバイト数を返します。

```
sizeof(変数または型)
```

のように使います。

```
char buffer[80];
```

ならば、sizeof(buffer) は 80 になります。sizeof(int) は 4 になります。

2.8.2 算術演算子

算術演算子とは、加減乗除のことです。ただし、整数型と浮動少数点型では意味合いが違ってくるものもあります。

1. + 足し算または符号
2. - 引き算または符号
3. * 掛け算
4. / 割り算
5. % 剰余

足し算、引き算、掛け算は整数型と浮動少数点型とも違いがありません。割り算は整数型では少数以下は切り捨てられます。剰余は、整数どおしの割り算の余りを求めます。浮動少数点型ではありません。

```
5 + 3 → 8
5 - 2 → 3
5 * 2 → 10
5 / 2 → 2
5 % 3 → 2
3.54 + 2.0 → 5.54
3.54 - 2.0 → 1.54
3.54 * 2.0 → 7.08
3.54 / 2.0 → 1.72
```

表 2.2　C における演算子の優先順位と結合規則

優先順位	演算子	式の結果	結合規則
高い	( )　関数呼び出し		左から右
	[ ] 配列添え字		左から右
	. ドット (構造体のメンバ)		左から右
	− > 矢印 (構造体のメンバ)		左から右
	!　論理否定	論理・算術	左から右
	〜 1 の補数	算術	左から右
	−　単項マイナス	算術	左から右
	++　インクリメント	算術	左から右
	−−　ディクリメント	算術	左から右
	&　アドレス		左から右
	*　乗算	算術	左から右
	/　除算	算術	左から右
	%　剰余 (割り算の余り)	算術	左から右
	+　加算	算術	左から右
	ー　減算	算術	左から右
	<<　左シフト	算術	左から右
	>>　右シフト	算術	左から右
	<　より小さい	論理	左から右
	<=　以下	論理	左から右
	>　より大きい	論理	左から右
	>=　以上	論理	左から右
	==　等しい	論理	左から右
	! =　等しくない	論理	左から右
	&　ビット毎の AND	算術	左から右
	^　ビット毎の EXOR	算術	左から右
	|　ビット毎の OR	算術	左から右
	& &　論理積	論理	左から右
	| |　論理和	論理	左から右
	? :　条件		左から右
	=　代入	算術	左から右
低い	,　カンマ		左から右

なお、整数型と浮動少数点型が混じった演算はキャストする必要があります。結果は、代入する変数の型になります。

2.8.3 インクリメンタリはデクリメンタリ演算子

1. + + インクリメンタリ演算子
2. - - デクリメンタリ演算子

以上 2 つがあります。変数の右か左につけて使われます。その場合の意味を下に示します。

```
j=i++;   /* j=i をやったあとで i=i+1 */
k=++i;   /* i=i+1 をやったあとで k=i */
j=i--;   /* j=i をやったあとで i=i-1 */
k=(--i); /* i=i-1 をやったあとで k=i */
```

2.8.4 bit 毎の論理演算子

1. & bit 毎の論理 AND
2. ^ bit 毎の論理 EXOR
3. | bit 毎の論理 OR
4. ~ bit 毎の論理 NOT（1 の補数)

bit 毎の論理演算子は以上があります。

bit 毎の論理 AND とは下に示すように j,i とも同じ位置の bit が両方とも 1 のとき 1 になり、そうでないときは 0 になります。

```
101101111 =j
111000101 =i
101000101 =j & i
```

bit 毎の論理 OR とは下に示すように j,i とも同じ位置の bit がどちらかが 1 のとき 1 になり、そうでないときは 0 になります。

```
100101101 =j
111000101 =i
111101101 =j | i
```

bit 毎の論理 EXOR とは下に示すように j,i とも同じ位置の bit が一致してないとき 1 のとき 1 になり、そうでないときは 0 になります。

```
100101101 =j
111000101 =i
011101000 =j ^ i
```

bit 毎の論理 NOT とは下に示すように j のそれぞれの bit が 1 なら 0 に、0 なら 1 にする。

```
100101101 =j
011010010 =~j
```

2.8.5　bit 毎のシフト演算子

1. >>　bit 毎の右シフト
2. <<　bit 毎の左シフト

シフト演算子は以上があります。

左シフトは指定したビット数だけ左にシフトします。開いた bit は 0 で埋められ、溢れた bit は捨てられます。

```
0100010001 =j
1000100010 =(j << 1)
0001000100 =(j << 2)
0010001000 =(j << 3)
```

右シフトは指定したビット数だけ右にシフトします。開いた bit は 0 で埋められ、溢れた bit は捨てられます。

```
0100010001 =j
0010001000 =(j >> 1)
0001000100 =(j >> 2)
0000100010 =(j >> 3)
```

2.8.6　複合演算子

1. +=
2. -=
3. *=
4. /=
5. %=
6. &=
7. ^=
8. |=
9. >>=
10. <<=

複合演算子には以上のものがあります。演算と代入を一緒にします。それぞれの意味は下に示すとおりです。

```
j+=i; /* j=(j + i) */
j-=i; /* j=(j - i) */
j*=i; /* j=(j * i) */
j/=i; /* j=(j / i) */
j%=i; /* j=(j % i) */
j&=i; /* j=(j & i) */
j|=i; /* j=(j | i) */
j^=i; /* j=(j ^ i) */
j>>=i; /* j=(j >> i) */
j<<=i; /* j=(j << i) */
```

2.8.7 条件演算子

1. >
2. <
3. > =
4. < =
5. ! =
6. = =

条件演算子は以上に示すもので意味は下に示します。

```
(j > i)  /* j が i より大きいなら真、そうでなければ偽 */
(j < i)  /* j が i より小さいなら真、そうでなければ偽 */
(j >= i) /* j が i 以上なら真、そうでなければ偽 */
(j <= i) /* j が i 以下なら真、そうでなければ偽 */
(j != i) /* j と i が異なっていれば真、そうでなければ偽 */
(j == i) /* j と i は等しければ真、そうでなければ偽 */
```

2.8.8 論理演算子

真偽の論理には以下のものがあります。

1. && 論理 AND
2. || 論理 OR

以下に使用例を示します。

```
  ((j > i) && (j == k)) /*(j > i) かつ (j == k) なら 真、そうでなければ偽*/
  ((j > i) || (j == k)) /*(j > i) または (j == k) なら 真、そうでなければ偽*/
```

上の方が論理 AND、下の方が論理 OR です。

2.9 ポインタ

ポインタは、簡単にいうと間接アドレッシングです。

```
char *s;/* char 型データを示すポインタ s の宣言 */
int *p; /* int 型データを示すポインタ p の宣言 */
struct seiseki *data;/* 構造体 seiseki 型データを示すポインタ data の宣言 */
```

このようにポインタは変数名の前に＊がつきます。

CPU は、メモリを番地で管理しています。何丁目何番地という感じです。変数は、中村さん、鈴木さん、佐藤さんといった具合です。

中村さんは、二丁目三番地なので、中村さんといわずに、二丁目三番地といってもいいわけですよね。

鈴木さんは、二丁目四番地なので、鈴木さんといわずに、二丁目四番地といってもいいわけですよね。

佐藤さんは、二丁目五番地なので、佐藤さんといわずに、二丁目五番地といってもいいわけですよね。

さて、郵便局では、二丁目三番地、二丁目四番地、二丁目五番地という箱があって仕分けされます。

いま、鈴木さんが転居しました。四丁目一番地です。そこで、郵便局は二丁目四番地の箱に四丁目一番地に転居したとメモをいれました。

これで、郵便局では、二丁目四番地の鈴木さんの郵便物をいれようとするとメモがあるので、メモをみて鈴木さんが四丁目一番地であることがわかります。そこで、四丁目一番地の箱に鈴木さんの郵便物をいれます。

ここで、メモを使うことが間接アドレッシングです。メモを書き換えるだけでどの箱にもいれることができます。

このメモの入った箱をポインタといいます。ポインタには郵便物は入れられません。メモだけです。

さて、話をもどしてポインタは変数のメモリアドレスです。しかし、変数は型によって使用するバイト数が異なります。

たとえば、鈴木さんには、広さんと佳子さんがすんでいることもあるし、佐藤さんは大家さんで、間借り人が 5 人いて、佐藤さんの住所を 6 人で共有することもあるので、一番地増やしてたら、一軒とも限りませんね。

たとえば、char なら 1 バイト、int なら 4 バイトであったりします。構造体なら何バイトでもあったりします。したがって、変数のアドレスと言っても使用するアドレス範囲があるので、ポインタは型で指定します。ですから、ポインタ p をインクリメントすると

```
p++;
```

p の示すアドレスの次ではなくて、p の型のメモリサイズ +1 になります。たとえば p が構造体で 30 バイトを使うものなら p の示すアドレスは 31 後ろになります。もし、そうでないと構造体の配列の場合不便ですよね。

このようにポインタの増やす値は、何番目という意味であって、アドレスそのものではありません。

2.9.1 const や void のポインタ

さて、ポインタは、型を指定しなくてはなりません。次の場合、ポインタ pa は const と書いてあるので定数という意味ではなくて、定数の整数型を示すポインタという意味です。

```
const int a[]={1,2,3,4,5,6,7,8,9,10};

const int *pa;/* const int 型データを示すポインタ pa の宣言 */
const int **ppa;/* const int 型データを示すポインタのポインタ ppa の宣言*/
```

定数型のデータを示すポインタはこのように const からはじめないといけません。

つぎの例は、文字列定数 s を示すポインタ p を使いますが、p は文字列定数型のポインタとして定義されています。

```
const char s[]="This is a pen.\0";
const char *p;
```

このように p は const char と宣言されていますね。

また、これは malloc のプロトタイプ宣言ですが

```
void *malloc(size_t size);
```

void *と書かれていますね。この場合、型指定なしのポインタであるということで、この関数の戻り値は、型としてはなんでもいいということです。あとでででてきます。

このように、ポインタでは const も void も意味が本来と違うので注意します。

2.9.2 ポインタのポインタ

さて、鈴木さんが、また引っ越しました。今度は六丁目三番地です。すると、郵便局は、二丁目四番地のメモを書き換えずに、四丁目一番地に六丁目三番地に転居したとメモをいれました。すると、鈴木さんに郵便がくると、二丁目四番地のメモから四丁目一番地に引っ越したとわかり、四丁目一番地のメモから六丁目三番地に転居したこととわかるので六丁目三番地に郵便物をいれます。こうすると、二丁目四番地宛の郵便物も、四丁目一番地宛の郵便物も、六丁目三番地に配達されますね。

この時、二丁目四番地のメモはメモのありかを示しているので、ポインタのポインタとよばれます。

```
#include <stdio.h>

const int a[]={1,2,3,4,5,6,7,8,9,10};

int main (void )
{

const int *pa;/* const int 型データを示すポインタ pa の宣言 */
```

```
const int **ppa;/* const int 型データを示すポインタのポインタ ppa の宣
言*/

    pa=a+5;  /* a[0] の 5 つ後ろのデータのアドレスを pa にセット */
    ppa=&pa; /* ppa に pa のアドレスをセット */
     printf("a[5]=%d *pa=%d **ppa=%d \n",a[5],*pa,**ppa);
    return 0;
}
```

実行結果

```
a[5]=6 *pa=6 **ppa=6
```

a[5] も*pa も**ppa も同じアドレスのデータ 6 を呼び出していますね。

なお、pa=a+5 は、pa,a も int 型であるので、4 バイトですので、+5 は実際は 4 倍された 20 が加算されます。このようにポインタの増やす値は、何番目という意味であって、数値そのものではありません。

2.9.3　&演算子

&演算子は、変数のアドレスを返します。

先の例では中村さんは、二丁目三番地なのでしたね。&演算子は、中村さんの住所を求めるものです。ですから

```
住所=&中村;/*住所には二丁目三番地が入ります。*/
```

となりますね。さて、

```
int i;
int *ptr;

  ptr=&i; /* i と*ptr は同じ */
  (*ptr)++; /* i++ と同じ  */
```

2.9.4　*演算子

*演算子は間接アクセスです。先の例では*演算子は、四丁目一番地のメモから鈴木さんを求めるものです。さて、

```
char i;

  *(&i)=*(&i)+3; /* i=i+3 は同じ */
```

2.9.5　関数へのポインタ

関数もメモリ上に配置されますから、番地があります。名前と番地は読み替えですから、関数の番地で呼び出すことができます。その時使うのが関数へのポインタです。

関数へのポインタが最も効果的なのは、ステートマシンへの実装です。

ステートマシンを if や switch で書くと

```
int old_state=1;
```

```
if (event){
 swicth(old_state)
   {
   case(state1):
     {
      文;
      old_state=state2;
      break;
     }
   case(state2):
     {
      文;
      old_state=state3;
      break;
     }
   case(state3):
     {
      文;
      old_state=state4;
      break;
     }
   case(state4):
     {
      文;
      old_state_state5;
      break;
     }
   case(state6):
     {
      文;
      old_state=state1;
      break;
     }
   }
}
```

すると、next　state の処理の前に if や switch の分岐が入ってオーバーヘッドが大きくなります。こういう時、関数へのポインタを使うと文末に次の state へのポインタに書き換えをおくと、次の起動でいきなり、次の state が始まるので、オーバーヘッドはありません。

したがって、状態遷移図に近いルーチンが書けます。これを if や switch で書くと噴水型プログラムになります。

関数へのポインタで先のステートマシンを書くと

```
void state1 (void); /* 関数のプロトタイプ宣言 */
void state2 (void); /* 関数のプロトタイプ宣言 */
void state3 (void); /* 関数のプロトタイプ宣言 */
void state4 (void); /* 関数のプロトタイプ宣言 */
void state5 (void); /* 関数のプロトタイプ宣言 */
void state6 (void); /* 関数のプロトタイプ宣言 */

void (*next)(void); /* 関数へのポインタ */

void state1 (void)  /* ステート 1 処理の関数 */
```

```
{
    文;
    next=state2;    /* 次ステートの記述 */
}
void state2 (void)  /* ステート 2 処理の関数 */
{
    文;
    next=state3;    /* 次ステートの記述 */
}
void state3 (void)  /* ステート 3 処理の関数 */
{
    文;
    next=state4;    /* 次ステートの記述 */
}
void state4 (void)  /* ステート 4 処理の関数 */
{
    文;
    next=state5;    /* 次ステートの記述 */
}
void state5 (void)  /* ステート 5 処理の関数 */
{
    文;
    next=state6;    /* 次ステートの記述 */
}
void state6 (void)  /* ステート 6 処理の関数 */
{
    文;
    next=state1;    /* 次ステートの記述 */
}

int main (void)
{
   next=state1;
   do
   {
    if(event){
      (*next)(); /* ポインタによる関数の呼び出し */
    }
   }
   while(loop_state);
```

この例では、条件判断でステートが変わるものではありませんでしたが、これに条件判断が加わると if や switch で書くと状態遷移図は程遠いぐちゃぐちゃの if や switch だらけになって手におえなくなります。

その点、関数へのポインタでは状態遷移図に忠実に書くことができます。

なお、関数へのポインタの配列によって表変換などにも使われます。表変換を if や switch で書くことは骨のおれる作業です。

表変換の例を示します。ここではアップダウンカウンタを表変換でシーケンスをくんでいます。

```
#include <stdio.h>
void state1 (void);/* 関数のプロトタイプ宣言 */
void state2 (void);/* 関数のプロトタイプ宣言 */
```

```
void state3 (void);/* 関数のプロトタイプ宣言 */
void state4 (void);/* 関数のプロトタイプ宣言 */
void state5 (void);/* 関数のプロトタイプ宣言 */
void state6 (void);/* 関数のプロトタイプ宣言 */

struct table_list  /* シーケンステーブルの構造体 */
{
void (*next)(void);/* 関数へのポインタ */
int next_state;    /*  次のシーケンス  */
};
/* カウントアップシーケンステーブル  */
static struct table_list table_up[6]={
{state1,1},{state2,2},{state3,3},
{state4,4},{state5,5},{state6,0}
};
/* カウントダウンシーケンステーブル  */
static struct table_list table_down[6]={
{state6,1},{state5,2},{state4,3},
{state3,4},{state2,5},{state1,0}
};

void state1 (void)
{     printf("state1\n");}
void state2 (void)
{     printf("state2\n");}
void state3 (void)
{     printf("state3\n");}
void state4 (void)
{     printf("state4\n");}
void state5 (void)
{     printf("state5\n");}
void state6 (void)
{      printf("state6\n");      }

int main (void)
{
 int  event=1;
 int i;
 int j=0;

   for (i=0;i<15;i++)
   {
    if(event){
     (*table_up[j].next)();
     j=table_up[j].next_state;
    }
   }
   for (i=0;i<15;i++)
   {
    if(event){
     (*table_down[j].next)();
     j=table_down[j].next_state;
    }
   }
   return 0;
```

```
}
```

2.10　#コマンド

#コマンドには先に書いた#include がすでに出てきました。この他に

1. #define
2. #undef
3. #ifdef
4. #if
5. #else
6. #endif

が、挙げられます。これらはプリプロセッサのコマンドです。コンパイルする前にソースを加工します。

2.10.1　#include

```
#include "ファイル名"
```

このように書くとこの位置にファイル名のファイルがまるまる読み込まれてコンパイルされます。したがって、同じような記述はファイルに落として#include すればいいのです。

```
#ifdef DEBUG
#include "show_all.h"
#endif
```

こうすると、DEBUG が#define されていると、show_all.h が読み込まれます。そうでないと読み込まれません。

2.10.2　#define

```
#define loop 10
```

こうすると loop は 10 になります。また、マクロというものがあります。

```
#define pi 3.1415926
#define area(x)   (pi*x*x)
```

こうすると area(x) は半径 x の円の面積になります。コンパイル前のソースの加工ですから、x に与えた値で式が展開されるだけです。計算されるのはコンパイラが計算して定数としてしまいます。

マクロは、使い方が難しいし, 副作用もありますので要注意です。

2.10.3 #undef

```
#define loop 10
#define loops 20
#undef loop
```

このように、一旦#define したものを無効にします。したがって loop は、なくなります。

2.10.4 #ifdef

#define されているかチェックします。

```
#ifdef DEBUG
DEBUG が定義されていたとき、ソースに挿入されるブロック
#endif
```

2.10.5 #if

```
#if 定数式
定数式が０でないときここに書かれたブロックがソースに挿入される
#else
定数式が０のときここに書かれたブロックがソースに挿入される
#endif
```

#endif は必要ですが、#else とその下のブロックはあってもなくてもよい。

2.10.6 #else

すでに、#if で説明しましたが、#ifdef でも同じです。

```
#ifdef DEBUG
DEBUG が定義されていたとき、ソースに挿入されるブロック
#else
DEBUG が定義されていないとき、ソースに挿入されるブロック
#endif
```

2.10.7 #endif

#if、#ifdef の終わりを意味します。

第 3 章

C でプログラムしてみる

C 言語は、普通は単独では使われず、標準ライブラリをくっつけて使われます。ここでは、標準ライブラリを使いながら標準ライブラリと C 言語を説明します。

さて、VineLinux5.0 にはマニュアルが、入っています。マニュアルを検索するやり方を説明します。

```
$ man 3 printf
```

これは、printf の使い方を調べるときに、man と入力して調べたい printf を分類 3 で調べるやり方です。マニュアルには分類があります。分類 3 はプログラミング関係です。すると、解説のページ末に関連項目が表示されます。

関連項目
printf(1), asprintf(3), dprintf(3), scanf(3), setlocale(3), wcrtomb(3), wprintf(3), locale(5)

たとえば、scanf(3) というのは、マニュアルの分類 3 にありますということです。また、printf(1) があります。これは、分類 1 にも printf があるということです。分類 1 はコマンド関係です。

このように、これ以降の説明は簡易の説明になっていますが、詳細は man コマンドで確認して下さい。

3.1 hello world!

C の入門書の最初のプログラムはこれです。

```
#include <stdio.h>

int main (void)
{
    printf("Hello world!\n"); /*文１*/
    return 0;                 /*文２*/
}
```

文 2 はもどり値を返す文ですね。さて、文 1 は、コンソールに”Hello world!”を書いて \n は改行せよということです。

3.1.1　printf

stdio.h

```
int printf(const char *format[, argument, ...]);
```

printf は書式指定つきコンソール出力です。

```
const char *format[, argument, ...]
```

const とされているので文字列 format は、文字列定数であることがわかりますね。

printf にはファイルに出力する fprintf、文字列に出力する sprintf、またほかに snprintf,vprintf,vfprinf,vsprintf,vsnprinf があります。

vprintf,vfprinf,vsprintf,vsnprinf は stdio.h だけではなく、stdarg.h をインクルードする必要があります。

これから説明するのは stdio.h に定義されている printf の機能であって、C 言語とは関係ないものなのですが C 言語の標準ライブラリとして定義されているものです。

ですから、標準ライブラリのないワンチップマイコン専用の C コンパイラでは提供されてないことも当たり前なのです。

さて書式は、表 3.1 に示します。

表 3.1　printf の書式

型指定	入力引数	出力の書式
数値		
d	整数	符号つき 10 進　int
i	整数	符号つき 10 進　int
o	整数	符号つき 8 進　int
u	整数	符号なし 10 進　int
x	整数	符号なし 16 進　int (a,b,c,d,e,f を使用)
X	整数	符号つき 10 進　int (A,B,C,D,E,F を使用)
f	浮動小数点数	符号つきの [-]dddd.dddd 形式の値
e	浮動小数点数	符号つきの [-]d.ddddе[+/-]ddd 形式の値
E	浮動小数点数	符号つきの [-]d.ddddE[+/-]ddd 形式の値
c	文字	1 つの文字
s	文字列ポインタ	文字列（ヌル文字に出会うか文字数制限まで)

たとえば、

```
#include <stdio.h>

int main (void)
{
int i=345;
float a=34.56;
```

```
    printf("i=%d a=%f\n",i,a);  /*文３*/
    return 0;
}
```

とすると、i=dddd a=dddd.dddd のように i=と a=が数値の前に表示されます。% のあとの文字は表 3.1 の形式です。

```
i=345 a=34.560001
```

となります。

```
#include <stdio.h>

int main (void)
{
int i=345;
float a=34.56;

    printf("i=%X a=%E\n",i,a);  /*文３*/
    return 0;
}
```

とすると、

```
i=159 a=3.456000E+01
```

と i は 16 進数と a は指数表示になりましたね。

表示の桁数を指定するには

```
#include <stdio.h>

int main (void)
{
int i=3456;
float a=34.56789;

    printf("i=%-7d a=%-8.3f\n",i,a); /*文３*/
    return 0;
}
```

このように、%-7d のように, マイナスの符号を含めて 7 桁表示、%-8.3f のようにマイナスの符号も含めて 8 桁表示で小数点以下は 3 桁などと指定します。

```
i=3456    a=34.568
```

となります。

3.1.2 書式の特殊文字

書式には特殊文字として表 3.2 に示すものがあります。

表 3.2　書式の特殊文字

書式の特殊文字	意味
\n	改行コード
\t	TAB コード
\b	バックスペースコード（BS)
\r	キャリッジリターンコード（CR)
\f	ラインフィード（LF)
\\	バックスラッシュ
\’	シングル・クオーテーション
\0	ヌル文字（文字列の終わりなどを示す時に使う)

3.2　argc,argv[]

プログラムになにかを渡して起動するとき、プログラムはいくつ何が渡されたかは、argc,argv[] で知ることができます。argc は、渡された数です。argv[] は渡されたものです。

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

int main (int argc, char *argv[])
{
 int i;
 int sum=0;
     /* 引数がないとき使い方を表示 */
     if (argc<2){
     printf("usage sum data ...\n");
     return 1;
     }else{
     /* 引数の数を表示 */
     printf("argc=%d \n",argc);
     /* 合計を求める */
     for(i=1;i<argc;i++){
      sum=sum+atoi(argv[i]);
     }
    printf("sum=%d \n",sum);
  }
  return 0;
}
```

このプログラムは、起動と共に渡されたデータの合計をもとめています。
argc はプログラム名も 1 と数えるので、データがないときは argc=1 となります。
char *argv[] は、後続する文字列（データですが）のアドレスの配列です。
実行すると、

1 から 10 まで渡したとき

```
$ ./sum 1 2 3 4 5 6 7 8 9 10
argc=11
sum=55
引数がないまま起動したとき
$ ./sum
usage sum data ...
```

このようになります。

3.2.1 atoi

stdlib.h

```
int atoi(char *string);
```

atoi は、ASCII の数字文字列を数値化して int で返します。atoi を含むライブラリ stdlib.h をインクルードする必要があります。

atoi は、認識できない文字が現れたところで変換を終了します。文字列が正しく変換されないと 0 を返します。また、オーバーフローについては考慮されていません。

3.3 関数と引数

関数に変数を渡して処理してもらうことがあります。呼び出した関数から、値をコピーするものを値渡しといい、ポインタでアドレスを渡してお互い同じ変数を共有する参照渡しがあります。

3.3.1 値渡し

値渡しはごく普通に行われます。特別なことをしなければ値渡しになります。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* 合計と数から平均を求める関数 */
float average (int kazu,int goukei)
{
   return ((float)goukei/(float)kazu);
}

int main (int argc, char *argv[])
{
 int i;
 int sum=0;
     /* 引数がないとき使い方を表示 */
     if (argc<2){
     printf("usage sum data ...\n");
     return 1;
     }else{
     /* 引数の数を表示 */
     printf("argc=%d \n",argc);
     /* 合計を求める */
```

```
    for(i=1;i<argc;i++){
     sum=sum+atoi(argv[i]);
    }
   printf("sum=%d average=%f \n",sum,average(argc-1,sum));
  }
  return 0;
}
```

このプログラムは先のプログラムに平均を求める関数 average を追加しました。この関数は、一番下の printf 文で呼び出されています。argc-1 が kazu として渡されています。このばあい、この 2 つの変数は別々で値がコピーされました。ここでは kazu は argc-1 が渡されたので argc と kazu は違う値です。それは argc=11 と kazu=10 ということです。

同じように sum が goukei として渡されています。このばあいも、この 2 つの変数は別々で値がコピーされました。

実行すると、

```
コンパイルするときは<math.h>をインクルードしているので
$gcc -Wall -o sum sum.c -lm
のようにリンクオプション-lm をつける。

１から 10 まで渡したとき
$./sum 1 2 3 4 5 6 7 8 9 10
argc=11
sum=55 average=5.500000
```

このようになります。

3.3.2　参照渡し

参照渡しでは、ポインタを使います。呼び出される関数に変数のアドレスを渡して、変数そのものを操作するのです。

```
#include <stdio.h>

/* ２つの数を入れ替える関数 */
void swap (int *i,int *j)
{
int temp;   /* 一時退避用変数 */

   temp=*i; /* i の示すアドレスに書かれてあるデータ値を temp にコピー*/
   *i=*j;   /* j の示すアドレスに書かれてあるデータ値を i の示すアドレス
に書く */
   *j=temp; /* temp の値を j の示すアドレスに書く */
}

int main (void)
{
 int j=25;
 int k=-33;

 printf("入れ替え前 j=%d k=%d \n",j,k);
 swap(&j,&k);
 printf("入れ替え後 j=%d k=%d \n",j,k);
```

```
 return 0;
}
```

この例では、main では、j,k を印刷しているだけでなにもしていません。ただ、swap を呼び出しているだけです。入れ替えは swap が行っています。そのため、swap には swap(&j,&k) という具合に j,k のアドレスを渡しているのです。

実行すると、

```
$ ./swap
入れ替え前 j=25 k=-33
入れ替え後 j=-33 k=25
```

このようになります。

3.4 関数と配列

関数に配列を渡すと、それは配列の先頭アドレスが渡されます。つまり、参照渡しなのです。とはいえ、ポインタであることは意識する必要はありません。ただの配列でいいのです。

```
#include <stdio.h>

void pdata(int data[])
{
  int j;
    for(j=0;j<5;j++){ /* 配列の印刷  */
          printf("data[%d]=%d \n",j,data[j]);
    }
}

int main (void)
{
int a[5]; /* 配列 a は main のなかにある自動変数 */
int i=0;
int x;

   for(x=0;x<5;x++){
         a[x]=(++i);
   }
   pdata(a);/* 配列 a の渡し */
   return 0;
}
```

このように渡せばいいのです。

すると、

```
$ ./a.out
data[0]=1
data[1]=2
data[2]=3
data[3]=4
data[4]=5
```

このように、a[] が data[] に渡されていますね。

では、data[] は a[] の参照渡しであることを示しましょう。

```
#include <stdio.h>

void pdata(int data[])
{
  int j;
    for(j=0;j<5;j++){ /* 配列の印刷  */
          printf("data[%d]=%d \n",j,data[j]);
          data[j]=data[j]+10; /* 配列の書き直し  */
    }
}

int main (void)
{
int a[5];
int i=0;
int x;

   for(x=0;x<5;x++){
         a[x]=(++i);
   }
   pdata(a);/* 配列を渡す */
    for(i=0;i<5;i++){      /* 配列の印刷  */
          printf("a[%d]=%d \n",i,a[i]);
    }
   return 0;
}
```

こうすると、a[] はかわりますね。ためしてみましょう。

```
$ ./a.out
data[0]=1
data[1]=2
data[2]=3
data[3]=4
data[4]=5
a[0]=11
a[1]=12
a[2]=13
a[3]=14
a[4]=15
```

a[] は pdata の関数によって書き換わっていますね。したがって、参照渡しですね。

さて、多次元はどうなるでしょう。たとえば、3 次元配列を渡すときは

```
#include <stdio.h>

void pdata(int data[][10][4]) /* 3 次元配列を受けとって印刷関数 */
{
  int j,k,l;
    for(j=0;j<5;j++){
      for(k=0;k<10;k++){
        for(l=0;l<4;l++){
          printf("data[%d][%d][%d]=%d \t",j,k,l,data[j][k][l]);
```

```
        }
        printf("\n");
      }
    }
}

int main (void)
{
 int a[5][10][4]; /* 3 次元配列 */
 int i=0;
 int x,y,z;

   for(x=0;x<5;x++){
     for(y=0;y<10;y++){
       for(z=0;z<4;z++){
         a[x][y][z]=(++i);
       }
     }
   }
   pdata(a);   /* 配列の渡し */
   return 0;
}
```

先ほどと同じですね。

では、受けとる側の次元サイズが違うときはどうするのでしょうか。

```
#include <stdio.h>

void pdata(int data[10][4]) /* 2 次元配列を受けとって印刷関数 */
{
  int k,l;
      for(k=0;k<10;k++){
        for(l=0;l<4;l++){
          printf("data[%d][%d]=%d \t",k,l,data[k][l]);
        }
        printf("\n");
      }
}

int main (void)
{
 int a[5][10][4]; /* 3 次元配列 */
 int i=0;
 int x,y,z;

   for(x=0;x<5;x++){
     for(y=0;y<10;y++){
       for(z=0;z<4;z++){
         a[x][y][z]=(++i);
       }
     }
   }
   for(x=0;x<5;x++){
      pdata(a[x]);   /* 3 次元から 2 次元配列の渡し */
   }
```

```
   return 0;
}
```

これは、3 次元配列を 2 次元に切り出して渡していますね。

3.4.1 ポインタと配列

ポインタと配列の関係を見てみましょう。

```
#include <stdio.h>
/* 1 から 10 までの配列 */
const int a[]={1,2,3,4,5,6,7,8,9,10};

int main (void )
{

const int *pa;/*  ポインタの宣言*/
int i;

    pa=a;    /* ポインタを初期化 */
    for(i=0;i<10;i++){
     printf("a[%d]=%d *(pa+%d)=%d \n",i,a[i],i,*(pa+i));
    };
    return 0;
実行結果
a[0]=1 *(pa+0)=1
a[1]=2 *(pa+1)=2
a[2]=3 *(pa+2)=3
a[3]=4 *(pa+3)=4
a[4]=5 *(pa+4)=5
a[5]=6 *(pa+5)=6
a[6]=7 *(pa+6)=7
a[7]=8 *(pa+7)=8
a[8]=9 *(pa+8)=9
a[9]=10 *(pa+9)=10
}
```

配列の値とポインタの示す値が同じになりましたね。ということは、配列はメモリに一列に並んでいることになります。したがって、1 次元配列では、ポインタと配列は表現が違うだけのものです。

では、2 次元配列はどうなっているのでしょう。

```
#include <stdio.h>
/* 1 から 10 までの配列 */
const int a[2][5]={{1,3,5,7,9},{2,4,6,8,10}};

int main (void )
{

const int *pa;/*  ポインタの宣言*/
int i;
int x,y;

    pa=a;    /* ポインタを初期化 */
```

```
    for(i=0;i<10;i++){
     x=i / 5;
     y=i % 5;
     printf("a[%d][%d]=%d *(pa+%d)=%d \n",x,y,a[x][y],i,*(pa+i));
    };
    return 0;
}
```

実行結果

```
a[0][0]=1 *(pa+0)=1
a[0][1]=3 *(pa+1)=3
a[0][2]=5 *(pa+2)=5
a[0][3]=7 *(pa+3)=7
a[0][4]=9 *(pa+4)=9
a[1][0]=2 *(pa+5)=2
a[1][1]=4 *(pa+6)=4
a[1][2]=6 *(pa+7)=6
a[1][3]=8 *(pa+8)=8
a[1][4]=10 *(pa+9)=10
}
```

同じように、並んでいますね。まず、a[0][0] から a[0][4]、つぎに、a[1][0] から a[1][4] という具合です。

したがって、

```
#include <stdio.h>
/* 1 から 10 までの配列 */
const int a[2][5]={{1,3,5,7,9},{2,4,6,8,10}};

int main (void )
{

const int *pa[2];/*  ポインタの宣言*/
int i;
int x,y;

    pa[0]=a[0];    /* ポインタを初期化 */
    pa[1]=a[1];    /* ポインタを初期化 */
    for(i=0;i<10;i++){
     x=i / 5;
     y=i % 5;
     printf("a[%d][%d]=%d *(pa[%d]+%d)=%d \n",x,y,a[x][y],x,y,*(pa[x]+y));
    };
    return 0;
}
```

実行結果

```
a[0][0]=1 *(pa[0]+0)=1
a[0][1]=3 *(pa[0]+1)=3
a[0][2]=5 *(pa[0]+2)=5
a[0][3]=7 *(pa[0]+3)=7
a[0][4]=9 *(pa[0]+4)=9
a[1][0]=2 *(pa[1]+0)=2
a[1][1]=4 *(pa[1]+1)=4
a[1][2]=6 *(pa[1]+2)=6
a[1][3]=8 *(pa[1]+3)=8
a[1][4]=10 *(pa[1]+4)=10
```

```
}
```

さて、ここまでくると、

```
#include <stdio.h>
/* 1 から 10 までの配列 */
const int a[2][5]={{1,3,5,7,9},{2,4,6,8,10}};

int main (void )
{

const int *pa[2];/*  ポインタの宣言*/
int i;
int x,y;

    pa[0]=a[0];    /* ポインタを初期化 */
    pa[1]=a[1];    /* ポインタを初期化 */
    for(i=0;i<10;i++){
     x=i / 5;
     y=i % 5;
     printf("a[%d][%d]=%d pa[%d][%d]=%d \n",x,y,a[x][y],x,y,pa[x][y]);
    };
    return 0;
}
実行結果
a[0][0]=1 pa[0][0]=1
a[0][1]=3 pa[0][1]=3
a[0][2]=5 pa[0][2]=5
a[0][3]=7 pa[0][3]=7
a[0][4]=9 pa[0][4]=9
a[1][0]=2 pa[1][0]=2
a[1][1]=4 pa[1][1]=4
a[1][2]=6 pa[1][2]=6
a[1][3]=8 pa[1][3]=8
a[1][4]=10 pa[1][4]=10
}
```

ポインタと配列は同じものであることがわかりますね。

3.5　関数と構造体、共用体

構造体は、値渡しと参照渡しがあります。

```
#include <stdio.h>

struct data{
int bangou;
int size;
};

void pdata (struct data a)
{
   printf("bangou=%d  size=%d \n",a.bangou,a.size);
}
```

```
int main (void)
{
 struct data s[3];
 int i;

   s[0].bangou=1;
   s[0].size=12;
   s[1].bangou=2;
   s[1].size=34;
   s[2].bangou=3;
   s[2].size=56;

   for(i=0;i<3;i++){
      pdata(s[i]);
   }
   return 0;
}
```

これは、関数 pdata に対して値渡しになります。したがって、関数 pdata で s[] を操作できません。メンバーは、a.size のように a のメンバーとなります。

構造体は配列と違って、要素毎にコピーする必要がありません。構造体としてコピーされます。したがって、s[1] が a に全メンバーごとコピーされるのです。

つぎに示すものは参照渡しです。

```
#include <stdio.h>

struct data{
int bangou;
int size;
};

void pdata (struct data *a)
{
   printf("bangou=%d  size=%d \n",a->bangou,a->size);
}

int main (void)
{
 struct data s[3];
 int i;

   s[0].bangou=1;
   s[0].size=12;
   s[1].bangou=2;
   s[1].size=34;
   s[2].bangou=3;
   s[2].size=56;

   for(i=0;i<3;i++){
      pdata(&s[i]);
   }
   return 0;
}
```

この場合、a はポインタです。メンバーは、a->size のように->を使います。参照渡しなので、関数 pdata から s 渡された s[] を操作できます。

では、参照渡しのメンバーのメンバーはどうなるのでしょう。

```
#include <stdio.h>

struct point{
int x;
int y;
};

struct data{
int bangou;
struct point size;
};

void pdata (struct data *a)
{
   printf("bangou=%d  size.x=%d size.y=%d\n",a->bangou,a->size.x,a->size.y);
}

int main (void)
{
 struct data s[2];
 int i;

   s[0].bangou=1;
   s[0].size.x=100;
   s[0].size.y=200;

   s[1].bangou=2;
   s[1].size.x=300;
   s[1].size.y=400;

   for(i=0;i<2;i++){
      pdata(&s[i]);
   }
   return 0;
}
```

上に示すように、a->size.x のように普通の構造体のメンバーでアクセスできます。実行すると、

```
$ ./a.out
bangou=1  size.x=100 size.y=200
bangou=2  size.x=300 size.y=400
```

このようになります。

3.6　動的確保領域

stdlib.h

プログラムでは、変数はプログラムを書いたとき宣言されています。しかし、実行時に変数を作る必要があることがあります。これを動的確保といいます。たとえば、ファイルからテキストを読み込むプログラムを考えてみれば、ファイルの大きさはプログラム時点ではわかりません。実行してファイルを読み込みながらメモリの大きさを広げてゆくことになります。そういうときに便利です。

3.6.1 malloc

stdlib.h

```
void *malloc(size_t size);
```

malloc は size バイトの領域を確保し、領域の先頭アドレスをポインタとして返します。void *と書かれていますが、malloc の返すポインタがどの型のポインタにもなれるということです。

たとえば、構造体のポインタを返す例をしましょう。

```
struct point{
int x;
int y;
};

struct sikaku {
struct point left;
struct point bottom;
};

struct sikaku *windows;

   windows=malloc(sizeof(struct sikaku)*4);
```

この場合、malloc は、sikaku 型のポインタを返すことになります。

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

int main (void)
{
struct point{ /* 座標の構造体 */
int x;
int y;
};

struct sikaku { /* 四角形の構造体  */
struct point left; /* 左上の座標  */
struct point bottom;/* 右下の座標 */
};
struct sikaku *windows; /* windows という構造体 sikaku のポインタ */
int i;
       /* windows という領域を 4 つ確保  */
       windows=(struct sikaku *)malloc(sizeof(struct sikaku)*4);
       /* 初期化  */
       windows[0].left.x=10;windows[0].left.y=20;
```

```
    windows[0].bottom.x=15;windows[0].bottom.y=25;
    windows[1].left.x=110;windows[1].left.y=120;
    windows[1].bottom.x=115;windows[1].bottom.y=125;
    windows[2].left.x=210;windows[2].left.y=220;
    windows[2].bottom.x=215;windows[2].bottom.y=235;
    windows[3].left.x=310;windows[3].left.y=320;
    windows[3].bottom.x=315;windows[3].bottom.y=325;
    /*  表示   */
    for(i=0;i<4;i++){
      printf("widows[%d].left.x=%d\t",i,windows[i].left.x);
      printf("widows[%d].left.y=%d \n",i,windows[i].left.y);
      printf("widows[%d].bottom.x=%d\t",i,windows[i].bottom.x);
      printf("widows[%d].bottom.y=%d \n",i,windows[i].bottom.y);
    }
  /* 領域の解放  */
  free((struct sikaku *)windows);
  return 0;
}
```

もちろん、いちいち書かなくてもエラーにはなりませんが、明示的に上のように書いた方が良いのです。

malloc は領域が確保できないとき NULL を返します。ここでは、そのルーチンを省略していますが、書いた方がよいでしょう。なお、size が 0 のときも NULL を返します。

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

int main (void)
{
int i;
int *s;
  /* 動的領域確保   */
  if(NULL==(s=(int *)malloc(sizeof(int)*100))){
    printf("領域が確保できませんでした。\n");
    return 1;
  }
  /*  領域を 1 から 100 の番号をつける  */
  for(i=0;i<100;i++){
    s[i]=i;
  }
  /*  番号の印刷 */
  for(i=0;i<100;i++){
    printf("s[%d]=%d \n",i,s[i]);
  }
  free((int *)s); /* 領域を解放する */
  return 0;
}
```

s は int 型の 100 個の領域を malloc からもらいます。

malloc で確保した領域はプログラムが終わる前に解放しなくてはなりません。free((int *)s) がそれにあたります。

3.6.2 calloc

stdlib.h

```
void *calloc(size_t nitimes,size_t size);
```

size バイトでしめたサイズのブロックを nitimes 個作り、ブロックを 0 で初期化します。ブロックの先頭をポインタに返します。確保した領域はプログラムが終わる前に解放しなくてはなりません。free(s) がそれにあたります。

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

int main (void)
{
int i;
int *s;
    /* int サイズのブロックを 100 個確保する*/
    if(NULL==(s=(int *)calloc(100,sizeof(int)))){
      printf("領域確保に失敗しました。\n");
      return 1;
    }
    for(i=0;i<100;i++){
       /* 初期化されていることの確認*/
       printf("s[%d]=%d \n",i,s[i]);
       s[i]=i;
       printf("s[%d]=%d \n",i,s[i]);
    }
    free((int *)s);
    return 0;
}
```

3.6.3 realloc

stdlib.h

```
void *realloc(void *block,size_t size);
```

block は、一旦 free で解放した以前領域確保したブロックをしめします。size は必要に応じて大きくしても小さくしても構いません。ただし、再確保した領域が前と違う場所になることがあります。再確保できないと NULL を返します。size が 0 の場合も NULL が返されます。block が NULL ポインタであると malloc と同じように作用します。

3.6.4 free

stdlib.h

```
void free(void *block);
```

すでに割り当てられている領域を解放します。free は malloc,calloc,realloc で確保した領域を解放します。

3.7 入出力

3.7.1 getchar

stdio.h

```
int getchar(void);
```

標準入力 (stdin) から 1 文字入力し int で返します。ファイルの終わりまたはエラーのとき EOF を返します。次の getc(stdin) と定義されています。

```
#include <stdio.h>

int main (void)
{
int c;

   do
   {
     c=getchar();
     printf("%c",c);
   }while(c != 'q');
   return 0;
}
```

3.7.2 getc

stdio.h

```
int getc(FILE *stream);
```

steam は、シーケンシャルアクセスの I/O です。ファイル、キーボード、パイプもこれにあたります。FILE は stdio.h で定義されている構造体です。FILE 構造体は、ファイルアクセスのための作業エリアの構造体です。したがって、ファイルを扱うときは、FILE 構造体のポインタを定義する必要があります。

では、stdin から読んでみましょう。

```
#include <stdio.h>

int main (void)
{
int c;

   while('q' != (c=getc(stdin)))
   {
     printf("%c",c);
   };
   printf("\n");
   return 0;
}
```

では、ファイルを読んでみましょう。

```
#include <stdio.h>

FILE *fp;

int main (void)
{
int c;
   if(NULL==(fp=fopen("getchar.c","r"))){
     printf("can not open getchar.c\n");
     return 1;
   };

   while(EOF !=(c=getc(fp)))
   {
     printf("%c",c);
   }
   printf("\n");
   return 0;
}
```

3.7.3 fgets

stdio.h

```
char *fgets(char *s,int size,FILE *stream);
```

fgets は FILE *stream から読み込み s が指すバッファに格納する。s の文字数は size-1 で制限される。読み込みは、改行文字か EOF を読み込んだあとで停止します。戻り値は s です。エラーや 1 文字も読み込んでないとき NULL を返します。

```
#include <stdio.h>
#define MAX_SIZE 80

FILE *rp;
char dest1[MAX_SIZE];
char dest2[MAX_SIZE];

int main (void)
{
char *s1;
char *s2;/* fgets の戻り値 */
int size=MAX_SIZE;
    /* file のオープン */
    if(NULL==(rp=fopen("fgets.c","r"))){
      printf("could not open fgets.c \n");
      return 1;
    }
    /*  ポインタの初期化  */
    s1=dest1;
    s2=dest2;
    /*  ファイルから読み込んで表示*/
    s2=fgets(s1,size,rp);
    while(s2 != NULL){
     printf("s1=%s \n",s1);
```

```
     printf("s2=%s \n",s2);
     s2=fgets(s1,size,rp);
    }
    /*  ファイルのクローズ  */
    fclose(rp);
    return 0;
}
```

3.7.4　putc

stdio.h

```
int putc(int c,FILE *stream);
```

文字 c を unsigned char にキャストして stream に書き込みます。戻り値は書き込まれた文字を int にキャストして、また、エラーが発生すると EOF を返します。

```
#include <stdio.h>

FILE *wp;
int main (void)
{
int c;
int i;

    /* ファイルのオープン */
    if(NULL==(wp=fopen("putc.txt","w"))){
      printf("could not open putc.txt \n");
      return 1;
    }
    /* A-Z まで書き込み */
    for(i=0;i<26;i++){
     c='A'+i;
     putc(c,wp);
    }
    /* 最後に改行の書き込み */
    putc('\n',wp);
    /* ファイルのクローズ */
    fclose(wp);
    return 0;
}
```

実行結果

```
$ cat putc.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

3.7.5　puts

stdio.h

```
int puts(const char *s);
```

puts は文字列を標準出力（コンソールなど）に出力します。puts の引数が、const となっていますので、変数はだめです。文字列定数のみに使えます。

```
#include <stdio.h>

int main (void)
{
const char *message[3]={
                        "Good morning\0",
                        "Good afternoon\0",
                        "Good evning\0"
                        };
int i;

  for (i=0;i<3;i++){
    puts(message[i]);
    };
  return 0;
}
```

3.7.6 putchar

stdio.h

```
int putchar(int c);
```

putc(c,stdout) と同じです。文字 c を stdout に出力します。詳細は man コマンドで参照して下さい。

3.7.7 scanf

stdio.h

```
int scanf(const char *format[, address, ....]);
```

scanf は標準入力ストリームから format に示される型で入力する。詳細は man コマンドで参照して下さい。

```
#include <stdio.h>

int main (void)
{
int a,b;

     printf("input a=");
     scanf("%d",&a);
     printf("input b=");
     scanf("%d",&b);
     printf("a+b=%d \n",a+b);
     return 0;
}
```

実行結果

```
input a=2
input b=3
a+b=5
```

3.7.8 sprintf

stdio.h

```
int sprintf(char *buffer,const char *format[, argument, ...]);
```

文字列 buffer に出力する printf です。詳細は man コマンドで参照して下さい。

3.8 ファイル入出力

FILE 構造体は、ファイルアクセスのための作業エリアの構造体です。したがって、ファイルを扱うときは、FILE 構造体のポインタを定義する必要があります。

3.8.1 fprintf

stdio.h

```
int fprintf(FILE *stream,const char *format[, argument, ...]);
```

printf と同じです。出力先が FILE であるだけの違いです。詳細は man コマンドで参照して下さい。

```
#include <stdio.h>

char buff[80];
FILE *rp;
FILE *wp;


int main (void)
{
char *s;
int i=1;
/* fprintf.c に行番号をつけて表示、fprintf.txt にコピーするプログラム */

  s=buff;
 /*  読み込みファイルのオープン*/
  if(NULL==(rp=fopen("fprintf.c","r"))){
   printf("could not open fprintf.c \n");
   return 1;
  }
  /*  書き込みファイルのオープン*/
  if(NULL==(wp=fopen("fprintf.txt","w"))){
   printf("could not open fprintf.txt \n");
   return 1;
  }
  /* 1 行よんで書き込みをファイルの最後まで */
  while(NULL != (fgets(s,80,rp))){
   printf("fprintf.c:%3d:%s",i,s);
   fprintf(wp,"fprintf.c:%3d:%s",i,s);
   i++;
  }
```

```
    /* ファイルのクローズ  */
    fclose(rp);
    fclose(wp);
    return 0;
}
```

実行結果

```
fprintf.c:  1:#include <stdio.h>
fprintf.c:  2:
fprintf.c:  3:char buff[80];
fprintf.c:  4:FILE *rp;
fprintf.c:  5:FILE *wp;
```

以下省略

3.8.2 fopen

stdio.h

```
FILE *fopen(const char *path,const char *mode);
```

fopen は、ファイルを開きます。読み書きいずれもファイルはオープンしなければなりません。path はファイル名を指定します。mode は読み書きのモードを指定します。詳細は man コマンドで参照して下さい。

```
#include <stdio.h>

char buff[80];
FILE *rp;
FILE *wp;


int main (void)
{
char *s;
/* fopen.c を fopen.txt にコピーするプログラム */

     s=buff;
   /*  読み込みファイルのオープン*/
    if(NULL==(rp=fopen("fopen.c","r"))){
     /* ファイルが開けないときメッセージ表示*/
     printf("could not open fopen.c \n");
     /* プログラムから抜ける */
     return 1;
    }
   /*  書き込みファイルのオープン*/
    if(NULL==(wp=fopen("fopen.txt","w"))){
     /* ファイルが開けないときメッセージ表示*/
     printf("could not open fopen.txt \n");
     /* プログラムから抜ける */
     return 1;
    }
   /* 1 行よんで書き込みをファイルの最後まで */
    while(NULL != (fgets(s,80,rp))){
     fputs(s,wp);
    }
```

```
    /* ファイルのクローズ  */
     fclose(rp);
     fclose(wp);
     return 0;
}
```

3.8.3 fputc

stdio.h

```
int fputc(int c,FILE *stream);
```

putc と同じ。　詳細は man コマンドで参照して下さい。

3.8.4 fputs

stdio.h

```
int fputs(char *s,FILE *stream);
```

puts は stdout でしたが、stream になっているだけです。ですから、puts は fputs(s,stdout) と同じです。詳細は man コマンドで参照して下さい。

```
#include <stdio.h>
const char *message[3]={
                        "Good morning\n",
                        "Good afternoon\n",
                        "Good evning\n"
                        };
FILE *wp;

int main (void)
{
int i;
  /* ファイルのオープン */
  if(NULL==(wp=fopen("fputs.txt","w"))){
    printf("could not open fputs.txt \n");
    return 1;
  }
  /* ファイルへの書き込み  */
  for (i=0;i<3;i++){
    fputs(message[i],wp);
    };
  /* ファイルのクローズ */
  fclose(wp);
  return 0;
}
```

実行結果

```
$ cat fputs.txt
Good morning
Good afternoon
Good evning
```

3.8.5 fgetc

stdio.h

```
int fgetc(FILE *stream);
```

getc が stdin に対して fgetc は stream になって拡張されたもの。詳細は man コマンドで参照して下さい。

3.8.6 fscanf

stdio.h

```
int fscanf(FILE *stream,const char *format[,address,...]);
```

scanf が stdin に対して fscanf は stream になって拡張されたもの。詳細は man コマンドで参照して下さい。

3.8.7 fread

stdio.h

```
size_t fread(void *ptr,size_t size,size_t n,FILE *stream);
```

詳細は man コマンドで参照して下さい。

3.8.8 fwrite

stdio.h

```
size_t fwrite(void *ptr,size_t size,size_t n,FILE *stream);
```

詳細は man コマンドで参照して下さい。

3.8.9 fseek

stdio.h

```
int fseek(FILE *stream,long offset,int whence);
```

詳細は man コマンドで参照して下さい。

3.9 文字列操作

文字列操作は、文字列のポインタを渡して使います。戻り値は、文字列のポインタになることもあります。

以下において、size_t という型がでてきますが、整数型です。コンパイラ依存のため、それぞれのライブラリで typedef で size_t を定義しています。

3.9.1　strcpy

string.h

```
char *strcpy(char *dest,const char *src);
```

strcpy は、文字列 dest に文字列 src をコピーします。文字列 src のコピーはヌル文字をコピーしたところで終わりになります。戻り値は、dest です。

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

const char src[]="コピー後こうなります。\0";
char a[80],b[80];

int main (void)
{
char *s; /* strcpy の戻り値  */
char *dest; /* コピー先  */

     s=a;
     dest=b;
     s=strcpy(dest,src);/* src を dest にコピー */
     printf("dest=%s \n",dest);
     printf("s=%s \n",s);
     return 0;
}
実行結果
dest=コピー後こうなります。
s=コピー後こうなります。
```

3.9.2　stpcpy

string.h

```
char *stpcpy(char *dest,const char *src);
```

文字列をコピーして、戻り値にコピーした文字列の最後をポインタとして返す。したがって、次のようにすると、2 つの文字列はつながります。なお、_GNU_SOURCE を定義しておく必要があります。

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>

const char src[]="つづきます。\0";
const char org[]="このあとに\0";
char s[80];

int main (void)
{
char *dest; /* stpcpy の戻り値  */
```

```
        dest=s;
        dest=stpcpy(dest,org);
        printf("s=%s \n",s);
        dest=stpcpy(dest,src);
        printf("s=%s \n",s);
        return 0;
}
```

実行結果

```
s=このあとに
s=このあとにつづきます。
```

3.9.3 strncpy

string.h

```
char *strncpy(char *dest,const char *src,size_t n);
```

strncpy は、strcpy と同じですが、最大でも n バイトに制限されます。

3.9.4 strchr

string.h

```
int strchr(const char *src,int c);
```

strchr は、文字列 src の中から文字 c を探し、複数ある時は最初に文字 c のあるところをポインタで返す。もし、見つからないときは、NULL を返す。

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

const char s1[]="This is a pen.\0";

int main (void)
{
const char *p1;
char *p2;
int c='i';

        p2=strchr(s1,c);
        if(p2==NULL){
         printf("not found(%c)\n",c);
         return 1;
        }
        printf("find(%c)\n",c);
        printf("%s \n",s1);
        for(p1=s1;p1<p2;p1++){
          printf(" ");
        };
        printf("↑\n");
        return 0;
}
```

実行結果

```
This is a pen.
```

```
      ↑
```

ここでは、戻り値の示すところに↑で示しています。

3.9.5 strrchr

string.h

```
int strrchr(const char *src,int c);
```

strrchr は、文字列 src の中から文字 c を探し、複数ある時は最後に文字 c のあるところをポインタで返す。もし、見つからないときは、NULL を返す。

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

const char s1[]="This is a pen.\0";

int main (void)
{
const char *p1;
char *p2;
int c='i';

     p2=strrchr(s1,c);
     if(p2==NULL){
      printf("not found(%c)\n",c);
      return 1;
     }
     printf("find(%c)\n",c);
     printf("%s \n",s1);
     for(p1=s1;p1<p2;p1++){
       printf(" ");
     };
     printf("↑\n");
     return 0;
}
```

実行結果

```
This is a pen.
      ↑
```

ここでは、戻り値の示すところに↑で示しています。

3.9.6 strpbrk

string.h

```
char *strpbrk(const char *src,const char *accept);
```

strpbrk は、文字列 src の中に文字列 accept に含まれる文字が見つかると、最初に見つかったその位置をポインタで返す。見つからなければ、NULL を返す。

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

```
const char s1[]="This is a pen.\0";/* 文字列  */
const char s2[]="abcdefg\0"; /* 探す文字の一覧 */
int main (void)
{
const char *p1;
char *p2;

     p2=strpbrk(s1,s2);
     if(p2==NULL){
      printf("not found(%s)\n",s2);
      return 1;
     }
     printf("find(%s)\n",s2);
     printf("%s \n",s1);
     for(p1=s1;p1<p2;p1++){
       printf(" ");
     };
     printf("↑\n");
     return 0;
}
```

実行結果

```
This is a pen.
        ↑
```

ここでは、戻り値の示すところに↑で示しています。

3.9.7 strcat

string.h

```
char *strcat(char *dest,const char *src);
```

strcat は、文字列 dest に文字列 src をくっつけて dest に返します。dest は十分な大きさがです。strcat の戻り値は dest です。

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

const char src[]="つづきます。\0";
const char org[]="このあとに\0";
char a[80],b[80];

int main (void)
{
char *s; /* strcpy の戻り値  */
char *dest; /* コピー先  */

     s=a;
     dest=b;
     s=strcpy(dest,org); /* org を dest にコピー */
     printf("dest=%s \n",dest);
     s=strcat(dest,src); /* dest に src をくっつける */
     printf("dest=%s \n",dest);
     printf("s=%s \n",s);
```

```
    return 0;
}
```

実行結果

```
dest=このあとに
dest=このあとにつづきます。
s=このあとにつづきます。
```

3.9.8　strncat

string.h

```
char *strncat(char *dest,const char *src,size_t n);
```

strncat は、strcat と同じですが、最大 n バイトに制限されます。

3.9.9　strcmp

string.h

```
int strcmp(const char *s1,const char *s2);
```

strcmp は文字列 s1,s2 を最初の文字から比較して、対応する文字が異なるか、文字列が終わるかするまで行われます。戻り値は

strcmp **が負**　s1 が s2 より小さいとき
strcmp **が** 0　s1 と s2 が等しいとき
strcmp **が正**　s1 が s2 より大きいとき

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

const char s1[]="abcde\0";
const char s2[]="abcd\0";

int main (void)
{
int i;
     i=strcmp(s1,s2);
     printf("s1=%s \n",s1);
     printf("s2=%s \n",s2);
     printf("strcmp=%d \n",i);
     return 0;
}
```

3.9.10　strncmp

string.h

```
int strncmp(const char *s1,const char *s2,size_t n);
```

strncmp は文字列 s1,s2 を最初の文字から比較して、対応する文字が異なるか、文字列が終わるか n バイトまで行われます。strcmp と違うところは n バイトという制限があるこ

とです。

3.9.11 strspn

string.h

```
size_t strspn(const char *s,const char *accept);
```

strspn は、文字列 s,accept の最初から一致する長さの文字数を返す。詳細は man コマンドを参照してください。

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

const char str1[]="abcdefghijkl\0";
const char str2[]="abcd1234";

int main (void)
{
size_t i;
const char *s1;
const char *s2;

     s1=str1;
     s2=str2;
     i=strspn(s1,s2);
     printf("s1=%s \n",s1);
     printf("s2=%s \n",s2);
     printf("strspn=%d \n",i);
     return 0;
}
```

実行結果

```
s1=abcdefghijkl
s2=abcd1234
strspn=4
```

3.9.12 strcspn

string.h

```
size_t strcspn(const char *s,const char *reject);
```

strcspn は文字列 s,reject の最初から不一致の長さの文字数を返します。詳細は man コマンドを参照してください。

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

const char str1[]="abcdefghijkl\0";
const char str2[]="efghi123";

int main (void)
{
size_t i;
```

```
const char *s1;
const char *s2;

     s1=str1;
     s2=str2;
     i=strcspn(s1,s2);
     printf("s1=%s \n",s1);
     printf("s2=%s \n",s2);
     printf("strcspn=%d \n",i);
     return 0;
}
```

実行結果

```
s1=abcdefghijkl
s2=efghi123
strcspn=4
```

3.9.13　strstr

string.h

```
char *strstr(const char *haystack,const char *needle);
```

strstr は、文字列 haystack のなかに文字列 needle がないかを探し、あれば、最初にあれわれる位置をポインタで返します。もし、なければ NULL を返します。

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

const char s1[]="This is a pen.\0";
const char s2[]="pen\0";

int main (void)
{
const char *p1;
char *p2;

     p2=strstr(s1,s2);
     if(p2==NULL){
      printf("not found(%s)\n",s2);
      return 1;
     }
     printf("find(%s)\n",s2);
     printf(" %s \n",s1);
     for(p1=s1;p1<p2;p1++){
       printf(" ");
     };
     printf("↑\n");
     return 0;
}
```

実行結果

```
find(pen)
 This is a pen.
           ↑
```

3.9.14 strtok

string.h

```
char *strtok(char *str,const char *delim);
```

strtok は、文字列 str から文字集合 delim で示される区ぎり文字で、単語を切り出します。最初は、str を指定しますが、続けて行うには str には NULL を指定します。切り出された単語の文字列のポインタが戻り値になり、終わりは NULL を返します。

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

char s1[]="This is a pen.\0";
const char s2[]=" .\0";/* 区切り文字はスペースとピリオド */
int main (void)
{
char *p;

     p=strtok(s1,s2);
     while(p!=NULL){
      printf("%s\n",p);
      p=strtok(NULL,s2);
     }
     return 0;
}
実行結果
This
is
a
pen
```

3.9.15 strdup

string.h

```
char *strdup(const char *s);
```

strdup は、malloc を使って (strlen(s)+1) バイトのコピー先の領域を確保して文字列 s のコピーをつくります。

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

const char src[]="コピー後こうなります。\0";

int main (void)
{
char *dest;

     dest=strdup(src);
     printf("dest=%s \n",dest);
     return 0;
```

```
}
```

3.9.16 strndup

string.h

```
char *strndup(const char *s,size_t n);
```

strdup は、malloc を使って最大で n バイトのコピー先の領域を確保して文字列 s のコピーをつくります。もし、s が n より長い場合、n 文字のみコピーされ終端のヌル文字が最後につけられます。

3.9.17 strlen

string.h

```
size_t strlen(const char *s);
```

strlen は文字列 s の長さを戻します。ヌル文字は含みません。

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

const char s1[]="abcde\0";

int main (void)
{
size_t i;
     i=strlen(s1);
     printf("s1=%s \n",s1);
     printf("strlen=%d \n",i);
     return 0;
}
```

3.10 メモリ操作

3.10.1 memcpy

string.h

```
void *memcpy(void *dest,const void *src,size_t n);
```

memcpy は、メモリ領域 src の先頭 n バイトをメモリ領域 dest にコピーする。戻り値は dest と同じ。

詳細は man コマンドで参照して下さい。

3.10.2 memccpy

string.h

```
void *memccpy(void *dest,const void *src,int c,size_t n);
```

memcpy は、メモリ領域 src の先頭 n バイトをメモリ領域 dest にコピーする。ただし、n バイトコピー中に文字 c が見つかるとそこでコピーを中止する。戻り値は c のある次のポインタを指す。見つからない場合 NULL を返す。

詳細は man コマンドで参照して下さい。

3.10.3 memchr

string.h

```
void *memchr(const void *s,int c,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.4 memcmp

string.h

```
int *memcmp(const void *s1,const void *s2,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.5 memmove

string.h

```
void *memmove(void *dest,const void *src,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.6 memset

string.h

```
void *memset(void *s,int c,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.7 mblen

string.h

```
int mblen(const char *s,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.8 mbstowcs

string.h

```
size_t mbstowcs(wchar_t *dest,const char *src,size_t n);
```

詳細は man コマンドで参照して下さい。

3.10.9　mbstowcs

string.h

```
int mbtowc(wchar_t *pwc,const char *src,size_t n);
```

詳細は man コマンドで参照して下さい。

3.11　算術関係

math.h

各関数の man コマンドを参照して下さい。なお、エラーが発生した場合の処理は

```
man 7 math_error
```

を見て下さい。

3.11.1　acos

math.h

```
double acos(double x);
```

$\arccos(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.2　asin

math.h

```
double asin(double x);
```

$\arcsin(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.3　atan

math.h

```
double atan(double x);
```

$\arctan(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.4　atan2

math.h

```
double atan2(double y,double x);
```

$\arctan(\frac{y}{x})$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.5 atof

math.h

```
double atof(const char *s);
```

文字列 s を double 型の数値に変換して返します。詳細は man コマンドで参照して下さい。

3.11.6 ceil

math.h

```
double ceil(double x);
```

x を下回らない整数値を返します。詳細は man コマンドで参照して下さい。

3.11.7 cos

math.h

```
double cos(double x);
```

$\cos(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.8 cosh

math.h

```
double cosh(double x);
```

$\cosh(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.9 exp

math.h

```
double exp(double x);
```

$\exp(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.10 fabs

math.h

```
double fabs(double x);
```

x の絶対値を返します。詳細は man コマンドで参照して下さい。

3.11.11　floor

math.h

```
double floor(double x);
```

x をこえない最小の整数値を返します。詳細は man コマンドで参照して下さい。

3.11.12　log

math.h

```
double log(double x);
```

$\ln(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.13　log10

math.h

```
double log10(double x);
```

$\log(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.14　pow

math.h

```
double pow(double x,double y);
```

x^y を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.15　pow10

math.h

```
double pow10(int p);
```

10^p を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.16　sin

math.h

```
double sin(double x);
```

$\sin(x)$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.17　sinh

math.h

```
double sinh(double x);
```

sinh(x) を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.18 sqrt

math.h

```
double sqrt(double x);
```

$\sqrt{x}$ を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.19 tan

math.h

```
double tan(double x);
```

tan(x) を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.11.20 tanh

math.h

```
double tanh(double x);
```

tanh(x) を計算して結果を返します。詳細は man コマンドで参照して下さい。

3.12 時間関係

time.h

ここでは、構造体 tm が利用されます。構造体 tm はライブラリで宣言されています。

```
struct tm {
   int tm_sec;         /* 秒 (0~59) ただし、閏秒のため 60 は許される */
   int tm_min;         /* 分 (0~59) */
   int tm_hour;        /* 時間 (0~23) */
   int tm_mday;        /* 日 (1~31) */
   int tm_mon;         /* 月 (1~12) */
   int tm_year;        /* 年 (1900 年からの通算年数*/
   int tm_wday;        /* 曜日 (0~6) */
   int tm_yday;        /* 年内通算日 (0~365) */
   int tm_isdst;       /* 夏時間 (正ならば有効であり、0 または負なら無
効) */
};
```

また、time_t という型も使われます。整数型です。

3.12.1 time

time.h

```
time_t time(time_t *t);
```

time 関数は、現在の時間を 1970 年 1 月 1 日（00:00:00　UTC) からの経過時間として秒でポインタ t で示されたメモリに格納し戻り値に返します。UTC はグリニッジ平均時（GMT) でなく協定世界時を表します。t が NULL なら、戻り値だけです。

3.12.2 ctime

time.h

```
char *ctime(const time_t *time);
```

ctime は time 秒の時間を文字列にして返します。ただし、UTC 時間です。

```
Sat May 29 12:15:32 2010
```

このように文字列にされます。文字列をいれるバッファは 26 バイト以上でなければならない。

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

int main  (void)
{
char s[30];
char *p;
time_t timer;

     p=s;
     time(&timer);/* 時間の取得 */
     p=ctime(&timer);
     printf("%s \n",p);
     return 0;
}
```

3.12.3 gmtime

time.h

```
struct tm *gmtime(const time_t *time);
```

gmtime は、time 秒を構造体 tm に変換します。ただし、協定世界時（UTC）です。

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

int main  (void)
{
char s[30];
char *p;
time_t timer;
struct tm *now_time;
     p=s;
     time(&timer);/* 時間の取得 */
```

```
        now_time=gmtime(&timer);
        printf("%d 秒 \n",now_time->tm_sec);
        printf("%d 分 \n",now_time->tm_min);
        printf("%d 時 \n",now_time->tm_hour);
        printf("%d 日 \n",now_time->tm_mday);
        printf("%d 月 \n",now_time->tm_mon+1);
        printf("%d 年 \n",now_time->tm_year+1900);
        printf("%d 曜日 \n",now_time->tm_wday);
        printf("%d 年内通算日 \n",now_time->tm_yday);
        printf("%d 夏時間 \n",now_time->tm_isdst);

        return 0;
    }
```

3.12.4 asctime

time.h

```
char *asctime(const struct tm *tblock);
```

asctime は、構造体 tm で表された時間を文字列に変換します。

```
Sat May 29 12:15:32 2010
```

このように変換します。

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

int main  (void)
{
char s[30];
char *p;
time_t timer;

     p=s;
     time(&timer);/* 時間の取得 */
     p=asctime(localtime(&timer));
     printf("%s \n",p);
     return 0;
}
```

3.12.5 localtime

time.h

```
struct tm *localtime(const time_t *timer);
```

localtime は、time 秒を構造体 tm に変換します。ただし、現地時です。

```
#include <stdio.h>
#include <time.h>
char *youbi[]={
"日","月","火","水","木","金","土"
```

```
};

int main  (void)
{
char s[30];
char *p;
time_t timer;
struct tm *now_time;
     p=s;
     time(&timer);/* 時間の取得 */
     now_time=localtime(&timer);
     printf("%d 年 ",now_time->tm_year+1900);
     printf("%d 月 ",now_time->tm_mon+1);
     printf("%d 日 ",now_time->tm_mday);
     printf("%s 曜日 ",youbi[now_time->tm_wday]);
     printf("%d 時 ",now_time->tm_hour);
     printf("%d 分 ",now_time->tm_min);
     printf("%d 秒 \n",now_time->tm_sec);
     printf("年内通算日 %d 日め \n",now_time->tm_yday);
     printf("%d 夏時間 \n",now_time->tm_isdst);

     return 0;
}
```

3.12.6　stime

time.h

```
int stime(time_t *tp);
```

詳細は man コマンドで参照して下さい。

3.12.7　mktime

time.h

```
time_t mktime(struct tm *tm);
```

詳細は man コマンドで参照して下さい。

3.12.8　difftime

time.h

```
double difftime(time_t time2,time_t time1);
```

詳細は man コマンドで参照して下さい。

3.13　文字関数

ctype.h

3.13.1 isalnum

ctype.h

```
int isalnum(int c);
```

c が英数字（A-Z,a-z,0-9) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.2 isalpha

ctype.h

```
int isalpha(int c);
```

c が英字（A-Z,a-z) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.3 isascii

ctype.h

```
int isascii(int c);
```

c の下位バイトが 0 から 127(0x0-0x7f) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.4 iscntrl

ctype.h

```
int iscntrl(int c);
```

c の下位バイトが制御文字 (0x0-0x1f) あるいは削除文字 (0x7f) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.5 isdigit

ctype.h

```
int isdigit(int c);
```

c の下位バイトが数字 (0-9) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.6 islower

ctype.h

```
int islower(int c);
```

c の下位バイトが英小字 (a-z) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.7　isupper

ctype.h

```
int isupper(int c);
```

c の下位バイトが英大字 (A-Z) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.8　isspace

ctype.h

```
int isupper(int c);
```

c の下位バイトがスペース、タブ、復帰、改行、垂直タブ、改ページ (0x20,0x09-0x0d) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

3.13.9　isxdigit

ctype.h

```
int isdigit(int c);
```

c の下位バイトが 16 進数字 (0-9,A-F,a-f) かどうかを調べます。そうであれば、0 以外、そうでなければ 0 を返します。

第 4 章

コンパイラはなにしているか

この章は、暫定的に入っているものです。追加するかけすか思案中です。

4.1 最適化

4.1.1 定数式の移動

コンパイラには最適化オプションがあります。最適化とは何かというと、

```
#define k 12
#define l 5

  int i,j;
  int sum=0;

    for(i=0;i<10;i++){
    sum=sum+i+k+l;
    }
```

このばあい、k+l は、定数なのでコンパイラは次のように

```
#define k 12
#define l 5

  int i;
  int sum=0;

    for(i=0;i<10;i++){
    sum=sum+i+17;
    }
```

定数としてできるものは定数に置き換えます。これを最適化といいます。

```
  int i,j;
  int sum=0;

    for(i=1;i<10;i++){
    sum=sum+i;
    i=i;
    j=5;
    }
```

このループにおいて j=5; は、ループにおいてまったくここにある必要はありません。こんな無駄な文が何回も実行されると遅くなります。そこで、コンパイラはこれを

```
int i,j;
int sum=0;

  for(i=1;i<10;i++){
  sum=sum+i;
  }
  j=5;
```

と、変更します。これも最適化といいます。

この他、無駄な文の i=i; 削除なども行います。ただ、無駄な文は実行時間調整などの意味もあるので、最適化されると困る場合もあります。

したがって、最適化には最適化のレベルを最適化オプションにて指定します。