

構造体 (資料)

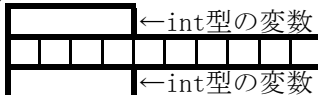
1. 構造体とは

1.1) 構造体の基本構造

構造体とは、いくつかの変数の集まりをひとまとめでしたものです。
配列との違いは、配列は同じデータ型の内容のみまとめて扱いますが、構造体は異なるデータ型をひとまとめでして扱うことができます。

・通常の変数や配列

```
int num;
char name[10];
int age;
```

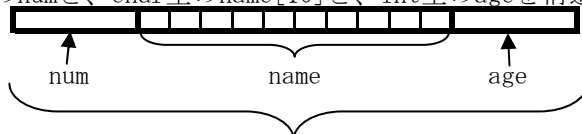


←int型の変数 ←char型の集合 ←int型の変数

各々の変数は、関連することなく個別に扱われます

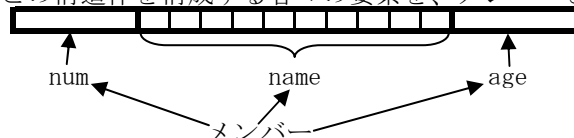
・構造体

int型のnumと、char型のname[10]と、int型のageを構造体にする

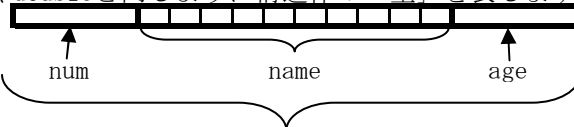


この内容はメモリー内に連続した領域として確保され、ひとまとまりとして扱います。

そして、この構造体を構成する各々の要素を、メンバーといいます。

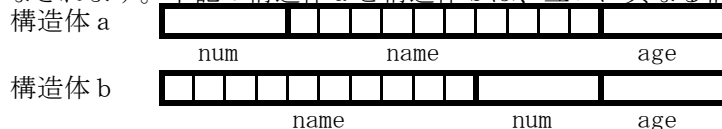


また、構造体を管理するために名前を付けることができます。この構造体に付けられた名前は、intやdoubleと同じように構造体の「型」を表します。



seito ← 構造体に付けた名前 = 型
この構造体は、seito型という型を持つことになります。

注意 構造体を構成するメンバーが同じでも、メンバーの順番が異なると異なる構造体であるとみなされます。下記の構造体 a と構造体 b は、互いに異なる構造体として扱われます。



1.2) なぜ構造体を利用するのか

構造体は、例えば次のような、ある一件分のデータを扱う処理に向いています。

成績データ							<div>int number; ひとり分のデータ (レコード) を構造体とすることで、番号～平均までの内容をひとまとまりとして扱うことができる。</div> <div>char name[10];</div> <div>int suugaku;</div> <div>int eigo;</div> <div>int goukei;</div> <div>double heikin;</div>
番号	名前	数学	英語	合計	平均		
10	浅野	100	91	191	95.5		
20	井上	80	97	177	88.5		
30	上野	98	85	183	91.5		
40	遠藤	86	93	179	89.5		

このように、複数の異なる型の要素を持つ内容をまとめて扱うことでプログラムの記述が簡便になります。

また、同じ構成を持つ構造体の変数間では、データのやり取りを一度に行うことができます。

構造体 a 10 | 浅野 | 100 | 91 | 191 | 95.5

構造体 b | | | | | |

b = a ; // 構造体の内容を一度に代入

構造体 a 10 | 浅野 | 100 | 91 | 191 | 95.5

構造体 b 10 | 浅野 | 100 | 91 | 191 | 95.5

2. 構造体の定義

2.1) 構造体の定義方法

次のプログラムは、ある学生のデーターを記憶するために、通常の変数と文字列を利用しています。

```
// kouTest01.c
: (define, include部省略)

/*
生徒のデータとして、クラス、学年、出席番号、名前、身長、体重を記録するために、
各々の内容について変数を用意した場合
*/
int main(void)
{
    int year;      // 学年
    int clas;      // クラス
    int number;    //出席番号
    char name[64]; // 名前
    double height; // 身長
    double weight; // 体重
    // データの設定
    year = 3;
    clas = 4;
    number = 18;
    strcpy(name, "田中");
    height = 168.2;
    weight = 72.4;
    // データの表示
    printf("%d\n", year);
    printf("%d\n", clas);
    printf("%d\n", number);
    printf("%s\n", name);
    printf("%f\n", height);
    printf("%f\n", weight);
    return(0);
}
```

このプログラムを、構造体を利用して書き直してみます。

- 構造体のタグ名

構造体は、必要に応じて複数個定義することができます。このとき、各々の構造体を識別するためにタグ名という名前をつけます。このタグ名は構造体を区別する名前であって、変数名ではありません。

- 構造体内のメンバー

構造体の中は構成要素である「メンバー」を定義します。メンバーにはint型などの変数を定義することができます。さらに、メンバーとして別の構造体型の変数を定義することもでき

```
// kouTest03.c
: (define, include部省略)
/*
生徒のデータとして、クラス、学年、出席番号、名前、身長、体重を記録するために、
構造体を用意した場合
*/

// student_tagという構造体を区別する名前「構造体タグ」（型ではない）を定義
struct student_tag
{
    int year;
    int clas;
    int number;
    char name[64];
    double height;
    double weight;
};

int main(void)
{
    struct student_tag data; // struct student_tag型を扱う変数としてdataを宣言
    // データの設定
    data.year = 3;
    data.clas = 4;
    data.number = 18;
    strcpy(data.name, "田中");
    data.height = 168.2;
    data.weight = 72.4;
    // データの表示
    printf("%d\n", data.year);
    printf("%d\n", data.clas);
    printf("%d\n", data.number);
    printf("%s\n", data.name);
    printf("%f\n", data.height);
    printf("%f\n", data.weight);
    return (0);
}

*****実行結果*****
3
4
18
田中
168.200000
72.400000
```

このプログラムでは構造体にstudent_tagという構造体タグ名を与え、year～weightまでの変数を構造体student_tagを構成するメンバーとして定義しています。

構造体の定義は単に構造体の構成を定義するだけです。メモリー内への領域の確保は行われません。

また構造体の定義では、最後の「}」の後ろに必ず「;」を記述してください。

2. 2) 構造体の型を持つ変数とメンバーの参照

- 変数の宣言

プログラム中では、次のように構造体struct student_tag型を持つ変数dataを宣言しています。

```
struct student_tag data; // struct student_tag型を扱う変数としてdataを宣言
```

この段階でメモリー上にstruct student_tag型の領域が確保され、変数名dataが与えられます。

- メンバーの参照

構造体の型を持つ変数dataでメンバーを参照する場合は、次のように「.」（ドット演算子）を利用します。

```
data.year = 3; // メンバーのint型変数year(学年)に3を記憶
data.clas = 4;
data.number = 18;
strcpy(data.name, "田中"); // メンバーのchar型変数name(氏名)に"田中"を記憶
data.height = 168.2;
data.weight = 72.4; // メンバーのdouble型変数weight(体重)に72.4を記憶
```

このように、「変数.メンバー名」と記述することで構造体内のメンバーを利用することができます。

- メンバー名と変数名

構造体変数dataのメンバー名と通常の変数名が重複していても、扱いが異なりますので共存することができます。

しかし、コードの可読性や保守性の観点からみれば、複数の同じ変数名が存在することはプログラムミスを誘発する原因となります。したがって、このような使い方は避けましょう。

```
// kouTest03a.c
//----- 一部省略
struct student_tag
{
    int year;
    int clas;
    int number;
    char name[64];
    double height;
    double weight;
};

int main(void)
{
    struct student_tag data; // struct student_tag型を扱う変数としてdataを宣言
    int year = 9999;
    double height = 123.456;
    // データの設定
    data.year = 3;
    data.height = 168.2;
    // データの表示
    printf("data.year=%d\n", data.year);
    printf("data.height=%f\n", data.height);
    printf("year=%d\n", year);
    printf("height=%f\n", height);
    return (0);
}

***** 実行結果 *****
data.year=3
data.height=168.200000
year=9999
height=123.456000
```

- ・メンバーへのキーボード入力

構造体のメンバーに対してキーボードからデーターを入力する場合、基本型の変数は先頭に&を付けて記述します。また、char型の配列は&を付けずに記述します。
次のプログラムは、構造体のメンバーyear、name、heightの内容をキーボードから入力したデーターで変更しています。

```
// kouTest04.c
:
main()のみを記載しています。
:
int main(void)
{
    struct student_tag data; // struct student_tag型を扱う変数としてdataを宣言
    // データの設定
    data.year = 3;
    data.clas = 4;
    data.number = 18;
    strcpy(data.name, "田中");
    data.height = 168.2;
    data.weight = 72.4;
    // データの変更
    printf("学年="); scanf("%d", &data.year);
    printf("氏名="); scanf("%s", data.name);
    printf("身長="); scanf("%lf", &data.height);
    // データの表示
    printf("%d\n", data.year);
    printf("%d\n", data.clas);
    printf("%d\n", data.number);
    printf("%s\n", data.name);
    printf("%f\n", data.height);
    printf("%f\n", data.weight);
    return (0);
}

*****実行結果*****
学年=4
氏名=佐々木
身長=180.4
佐々木
180.400000
72.400000
```

- 同じ構造体型を持つ変数間の代入
次のプログラムでは構造体を扱う変数としてdata1、data2を宣言し、data1に記憶されている構造体の内容をdata2に代入しています。

```
// kouTest05.c
: (define, include部省略)
int main(void)
{
// 構造体は、構造体の変数を、通常の変数のように利用できます
struct student_tag
{
    int year;
    int clas;
    int number;
    char name[64];
    double height;
    double weight;
};

struct student_tag data1; // struct student_tag型を扱う変数としてdata1を宣言
struct student_tag data2; // struct student_tag型を扱う変数としてdata2を宣言
// struct student_tag data1, data2; // これでもOK
// データの設定
data1.year = 3;
data1.clas = 4;
data1.number = 18;
strcpy(data1.name, "田中");
data1.height = 168.2;
data1.weight = 72.4;
// データのコピー
data2 = data1;
// データの表示
printf("*****data1*****\n");
printf("year=%d\n", data1.year);
printf("class=%d\n", data1.clas);
printf("number=%d\n", data1.number);
printf("name=%s\n", data1.name);
printf("height=%f\n", data1.height);
printf("weight=%f\n", data1.weight);
printf("*****data2*****\n");
printf("year=%d\n", data2.year);
printf("class=%d\n", data2.clas);
printf("number=%d\n", data2.number);
printf("name=%s\n", data2.name);
printf("height=%f\n", data2.height);
printf("weight=%f\n", data2.weight);
return (0);
}

*****実行結果*****
*****data1*****
year=3
class=4
number=18
name=田中
height=168.200000
weight=72.400000
*****data2*****
year=3
class=4
number=18
name=田中
height=168.200000
weight=72.400000
```

このように、同じ形を持つ構造体型の変数は、通常の変数と同じように互いに代入することができます。

3. 構造体の定義とtypedef

3.1) typedefによる新しい型の定義

これまでのプログラムでは、構造体はstruct student_tagという型を持つと紹介してきました。しかし、変数を宣言する際に、常にstruct student_tag型を記述するのでは面倒です。そこで、より簡潔に宣言するために新しい型を独自に定義するという方法が利用できます。つまり、従来のstruct student_tag型のように、構造体であることとタグ名を常に併記するのではなく、このstruct student_tag型を別の名前の型として利用できるようにしようということなのです。長い名前を短く愛称で呼ぶようなイメージです。

次のプログラムでは、今までのstruct student_tag型の代わりにseito型を定義しています。

```
// kouTest06.c
: (define, include部省略)
int main(void)
{
    // 構造体として、student_tagという構造体タグを宣言する
    struct student_tag
    {
        int year;
        int clas;
        int number;
        char name[64];
        double height;
        double weight;
    };

    // struct student_tag型を、別にseito型として定義する
    typedef struct student_tag seito;

    seito data; // 新しく定義されたseito型を扱う変数としてdataを宣言
```

このように、typedefによりstruct student_tag型と同じ意味を持つ型としてseito型を定義しています。この定義により、構造体を持つ変数dataを宣言する場合に、「struct student_tag data;」と書くのではなく、「seito data;」と記述できるようになります。

さらに、次のように構造体と型の宣言をまとめることもできます。

```
// kouTest07.c
: (define, include部省略)
int main(void)
{
    typedef struct student_tag
    {
        int year;
        int clas;
        int number;
        char name[64];
        double height;
        double weight;
    } seito;
```

構造体、タグ名、型を同時に宣言しています。

3.2) タグ名の省略

このように、構造体と型名が同時に宣言できるようになったことで、タグ名の役割は事実上、希薄になってしまいました。そこで、このタグ名も省略することができるようになっていま

次のプログラムでは、構造体と型を同時に宣言しタグ名を省略しています。

```
// kouTest08.c
: (define, include部省略)
int main(void)
{
    // さらに、student_tagという構造体タグ名を省略することもできる
    typedef struct
    {
        int year;
        int clas;
        int number;
        char name[64];
        double height;
        double weight;
    } seito;
```

タグ名を省略

このように、構造体の宣言は構造体のメンバーと型をtypedefで定義すればよいだけです。

3.3) 構造体の宣言場所

構造体はあるひとつの関数の中だけで利用されることは少なく、一般的には多くの関数で利用されます。このため、構造体の宣言は静的記憶域の適用範囲に宣言するのが一般的です。

次のプログラムでは、構造体をファイル有効範囲で宣言しています。

```
// kouTest09.c
: (define, include部省略)
typedef struct
{
    int year;
    int clas;
    int number;
    char name[64];
    double height;
    double weight;
} seito;

int main(void)
{
    seito data;
    // データの設定
    data.year = 3;
    data.clas = 4;
    data.number = 18;
    strcpy(data.name, "田中");
    data.height = 168.2;
    data.weight = 72.4;
    // データの表示
    printf("year=%d\n", data.year);
    printf("class=%d\n", data.clas);
    printf("number=%d\n", data.number);
    printf("name=%s\n", data.name);
    printf("height=%f\n", data.height);
    printf("weight=%f\n", data.weight);
    return (0);
}

*****実行結果*****
year=3
class=4
number=18
name=田中
height=168.200000
weight=72.400000
```

ファイル有効範囲で構造体を宣言

4. 構造体と関数

関数内で処理するデータは関数の引数で渡すことができますが、構造体も引数で渡すことができます。

4.1) 引数としての構造体

次のプログラムでは、構造体のseito型を持つ変数dataの内容を表示するために関数student_print()を呼び出しています。このとき、関数student_print()を呼び出す際の実引数として構造体型の変数を指定しています。関数側では、仮引数として宣言する変数はseito型で宣言します。これは、intなどの変数を受け渡しする場合に実引数と仮引数の型をあわせる記述と同じです。

```
// kouTest10.c
: (define, include部省略)
typedef struct
{
    int year; /* 学年 */
    int clas; /* クラス */
    int number; /* 出席番号 */
    char name[64]; /* 名前 */
    double height; /* 身長 */
    double weight; /* 体重 */
} seito;

void student_print(seito data);

int main(void)
{
    seito data;

    data.year = 3;
    data.clas = 4;
    data.number = 18;
    strcpy(data.name, "田中");
    data.height = 168.2;
    data.weight = 72.4;

    student_print(data); /* 呼び出し */
    return 0;
}

void student_print(seito data)
{
    printf("[学年]:%d\n", data.year);
    printf("[クラス]:%d\n", data.clas);
    printf("[出席番号]:%d\n", data.number);
    printf("[名前]:%s\n", data.name);
    printf("[身長]:%f\n", data.height);
    printf("[体重]:%f\n", data.weight);
    return;
}

*****実行結果*****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:168.200000
[体重]:72.400000
```

仮引数は、seito型で宣言する

4.2) 値渡しとしての構造体

関数に渡された変数は、値がコピーされて関数に渡されます。これを、値渡しといいます。引数に構造体の変数が指定された場合も、同じように値渡しとなります。次のプログラムでは、関数の内部で渡されたデータの内容を変更して表示しています。しかし、構造体のデータは、値渡しで関数に渡されますので呼び出しもとの構造体の内容には変化はありません。

```
// kouTest11.c
typedef struct
{
    int year; /* 学年 */
    int clas; /* クラス */
    int number; /* 出席番号 */
    char name[64]; /* 名前 */
    double height; /* 身長 */
    double weight; /* 体重 */
} seito;
void seito_print(seito data);
int main(void)
{
    seito data; // 構造体のseito型の変数、dataを宣言
    data.year = 3;
    data.clas = 4;
    data.number = 18;
    strcpy(data.name, "田中");
    data.height = 168.2;
    data.weight = 72.4;
    seito_print(data); /* 呼び出し */
    printf("**** 関数の呼出し後のデータ ****\n");
    printf("[学年]:%d\n", data.year);
    printf("[クラス]:%d\n", data.clas);
    printf("[出席番号]:%d\n", data.number);
    printf("[名前]:%s\n", data.name);
    printf("[身長]:%f\n", data.height);
    printf("[体重]:%f\n", data.weight);
    return (0);
}

// 呼び出した関数の中で、データを変更してみる
void seito_print(seito x)
{
    x.height = 99999.9999; // 身長を変更してみる
    printf("**** 関数の中で変更されたデータ(身長) ****\n");
    printf("[学年]:%d\n", x.year);
    printf("[クラス]:%d\n", x.clas);
    printf("[出席番号]:%d\n", x.number);
    printf("[名前]:%s\n", x.name);
    printf("[身長]:%f\n", x.height);
    printf("[体重]:%f\n", x.weight);
    return;
}

*****実行結果*****
**** 関数の中で変更されたデータ(身長) ****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:99999.999900
[体重]:72.400000
**** 関数の呼出し後のデータ ****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:168.200000
[体重]:72.400000
```

4.3) 構造体をポインタで参照する

構造体自身も、他の変数と同じようにポインタを利用して扱うことができます。

次のプログラムでは、seito型を扱うポインタ変数としてpdataを宣言して利用しています。

```
// kouTest12.c
typedef struct
{
    int year; /* 学年 */
    int clas; /* クラス */
    int number; /* 出席番号 */
    char name[64]; /* 名前 */
    double height; /* 身長 */
    double weight; /* 体重 */
} seito;

void seito_print(seito data);

int main(void)
{
    seito data; // 構造体のseito型の変数、dataを宣言
    seito *pdata; // seitoを扱うポインタ型の変数pdataを宣言
    pdata = &data; // pdataに構造体変数dataのアドレスを記憶する

    // *pdataで通常の変数と同じ扱いになるが、「.」が優先的に判断されるので()を付けておく
    (*pdata).year = 3;
    (*pdata).clas = 4;
    (*pdata).number = 18;
    strcpy( (*pdata).name, "田中" );
    (*pdata).height = 168.2;
    (*pdata).weight = 72.4;

    printf("[学年]:%d\n",      (*pdata).year);
    printf("[クラス]:%d\n",    (*pdata).clas);
    printf("[出席番号]:%d\n",  (*pdata).number);
    printf("[名前]:%s\n",      (*pdata).name);
    printf("[身長]:%f\n",      (*pdata).height);
    printf("[体重]:%f\n",      (*pdata).weight);
    return (0);
}

*****実行結果*****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:168.200000
[体重]:72.400000
```

ポインタで構造体を扱う場合には、通常の変数と同じように扱うことができる部分と、注意を要する部分があります。
ポインタを利用している部分を抜き出してみよう。

```
int main(void)
{
    seito data; // 構造体のseito型の変数、dataを宣言
    seito *pdata; // seitoを扱うポインタ型の変数pdataを宣言

    pdata = &data; // pdataに構造体変数dataのアドレスを記憶する

    // *pdataで通常の変数と同じ扱いになるが、「.」が優先的に判断されるので()を
    (*pdata).year = 3;
```

seito data;

ここでは、通常の変数と同じように、構造体seito型を持つ変数dataを宣言しています。

seito *pdata;

構造体変数seitoが記憶されているアドレスを記憶するための変数（ポインタ変数）pdataを宣言しています。変数pdataの型は、構造体seitoを扱いますので、seito型となります。

pdata = &data;

構造体dataが記憶されている先頭アドレスを、ポインタ変数pdataに記憶します。

このとき、dataの前にアドレス演算し&を付けることに注意してください。

(*pdata).year = 3;

*pdataは、pdataに記憶されているアドレスが示す内容（この場合は構造体data）となります。その構造体dataのメンバーを指定しますから、(*pdata).yearと記述します。

*pdata.yearと記述しないのは、.演算子の方が*よりも優先して解釈されてしまうためです。このため、(*pdata)で構造体を先に示し、次にその構造体のメンバーyearを指定していま

4.4) 構造体とアロー演算子 1

構造体のメンバーを指定するために、その都度、(*pdata).yearのように指定していたのではとても記述が面倒になります。そこで、このように()で括らなくても解釈させるための演算子として、アロー演算子(arrow operator)が用意されています。

アロー演算子によるメンバー指定は、次のような形式で記述します。

ポインタ変数->メンバー この「->」がアロー演算子です。

記述例 pdata->year = 10;
 x = pdata->number;

次のプログラムは、先ほどのプログラムをアロー演算子を利用して書き換えたものです。

```
// kouTest13.c
typedef struct
{
    int year; /* 学年 */
    int clas; /* クラス */
    int number; /* 出席番号 */
    char name[64]; /* 名前 */
    double height; /* 身長 */
    double weight; /* 体重 */
} seito;

void seito_print(seito data);

int main(void)
{
    seito data; // 構造体のseito型の変数、dataを宣言
    seito *pdata; // seitoを扱うポインタ型の変数pdataを宣言

    pdata = &data; // pdataに構造体変数dataのアドレスを記憶する
    pdata->year = 3;
    pdata->clas = 4;
    pdata->number = 18;
    strcpy(pdata->name, "田中");
    pdata->height = 168.2;
    pdata->weight = 72.4;

    printf("[学年]:%d\n", pdata->year);
    printf("[クラス]:%d\n", pdata->clas);
    printf("[出席番号]:%d\n", pdata->number);
    printf("[名前]:%s\n", pdata->name);
    printf("[身長]:%f\n", pdata->height);
    printf("[体重]:%f\n", pdata->weight);
    return (0);
}

*****実行結果*****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:168.200000
[体重]:72.400000
```

このように、ソースコードがすっきりとなりました。

4.5) 構造体をポインタで関数に渡す

関数の引数として変数のポインタ(アドレス)を渡すことができました。そして、関数の中では、受け取ったポインタの参照先の内容を変更することができました。つまり、呼出しもとの変数の内容を関数内で変更することができました。構造体の場合も、同じように構造体変数のポインタを関数に渡し、関数内で構造体の内容を変更することができます。次のプログラムでは、構造体のポインタを関数seito_print()に渡し、関数内で身長を変更しています。そして、関数の呼出し後に構造体のデーターを表示しています。

```
// kouTest15.c
: (define, include部省略)
typedef struct
{
    int year; // 学年
    int clas; // クラス
    int number; // 出席番号
    char name[64]; // 名前
    double height; // 身長
    double weight; // 体重
} seito;

void seito_print(seito *pdata); // seito型のポインタを受け取る

int main(void)
{
    seito data;

    seito *pdata;
    pdata = &data;

    data.year = 3;          data.clas = 4;
    data.number = 18;       strcpy(data.name, "田中");
    data.height = 168.2;    data.weight = 72.4;

    seito_print(pdata); // 構造体のアドレスを渡す

    printf("**** 関数を呼び出した後でデータを表示してみる ****\n");
    printf("[学年]:%d\n",    pdata->year); // -> でアクセスしてみる
    printf("[クラス]:%d\n",  pdata->clas);
    printf("[出席番号]:%d\n", pdata->number);
    printf("[名前]:%s\n",    pdata->name);
    printf("[身長]:%f\n",    pdata->height);
    printf("[体重]:%f\n",    pdata->weight);

    return (0);
}

void seito_print(seito *x)
{
    x->height = 99999.9999; // 受け渡されたデータを変更してみる
    printf("**** 関数内でデータを書き換えてみる ****\n");
    printf("[学年]:%d\n",    x->year); // ポインタ方で渡された構造体なので、->でアクセスする
    printf("[クラス]:%d\n",  x->clas);
    printf("[出席番号]:%d\n", x->number);
    printf("[名前]:%s\n",    x->name);
    printf("[身長]:%f\n",    x->height);
    printf("[体重]:%f\n",    x->weight);
    return;
}

*****実行結果*****
**** 関数内でデータを書き換えてみる ****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:99999.999900
[体重]:72.400000
**** 関数を呼び出した後でデータを表示してみる ****
[学年]:3
[クラス]:4
[出席番号]:18
[名前]:田中
[身長]:99999.999900
[体重]:72.400000
```

実行してみると、関数の中で書き換えられた身長⁵の値が呼び出し元の構造体の身長として表示されています。関数が受け取ったポインタ変数 `x` には構造体のアドレスがポインタに記憶されているため、関数内でメンバーの内容を変更すると呼び出しもとの構造体のメンバー内容を変更することになるからです。では、構造体を関数に渡す場合には変数による値渡しで行うと安全なのでしょうか。確かに値渡しで行うともとの内容は保護されますので安心です。しかし、実際のプログラムではポインタ渡しがよく利用されます。それは、次のような理由があるからです。

- 構造体自身のサイズが大きい場合(サイズの大きなメンバーが存在したり非常に多くのメンバーを持つ場合)、この内容をコピーするために時間がかかります。一方で、ポインタのアドレス値を渡すだけなら構造体のサイズに係わらず時間は短時間で済みます。
- 構造体を入れ子となった関数の引数として次々に受け渡す場合、関数の呼出しごとに通常の変数と同じように構造体の内容もスタックメモリー内に記憶されていきます。このため、構造体が大きなサイズであるほど、どんどんスタックメモリーを消費していくことに

5. 構造体と配列

構造体も配列を利用することができます。次のプログラムは、seito型を持つ配列data[]を利用しています。

```
// kouTest16.c
: (define, include部省略)
/* 構造体も配列を作成することができます。 */
typedef struct
{
    int year; // 学年
    int clas; // クラス
    int number; // 出席番号
    char name[64]; // 名前
    double height; // 身長
    double weight; // 体重
} seito;

void seito_print(seito data[], int n);

int main(void)
{
    seito data[5]; // 5件の構造体を記憶する配列data[]
    int n = 1;

    data[0].year = 3;      data[0].clas = 4;
    data[0].number = 18;   strcpy(data[0].name, "田中");
    data[0].height = 168.2; data[0].weight = 72.4;
    //
    data[1].year = 2;      data[1].clas = 1;
    data[1].number = 10;   strcpy(data[1].name, "佐々木");
    data[1].height = 160.5; data[1].weight = 58.5;
    // 3件目以降は省略

    seito_print(data, n); // 構造体の配列を渡す

    printf("**** 関数を呼び出した後でデータを表示してみる ****\n");
    printf("[学年]:%d\n", data[n].year);
    printf("[クラス]:%d\n", data[n].clas);
    printf("[出席番号]:%d\n", data[n].number);
    printf("[名前]:%s\n", data[n].name);
    printf("[身長]:%f\n", data[n].height);
    printf("[体重]:%f\n", data[n].weight);
    return (0);
}

void seito_print(seito x[], int n)
{
    x[n].height = 99999.9999; // 受け渡されたデータを変更してみる
    printf("**** 関数内でデータを書き換えてみる ****\n");
    printf("[学年]:%d\n", x[n].year);
    printf("[クラス]:%d\n", x[n].clas);
    printf("[出席番号]:%d\n", x[n].number);
    printf("[名前]:%s\n", x[n].name);
    printf("[身長]:%f\n", x[n].height);
    printf("[体重]:%f\n", x[n].weight);
    return;
}

*****実行結果*****
**** 関数内でデータを書き換えてみる ****
[学年]:2
[クラス]:1
[出席番号]:10
[名前]:佐々木
[身長]:99999.999900
[体重]:58.500000
**** 関数を呼び出した後でデータを表示してみる ****
[学年]:2
[クラス]:1
[出席番号]:10
[名前]:佐々木
[身長]:99999.999900
[体重]:58.500000
```


実行すると、配列の場合と同じように関数に渡された配列の内容を書き換えると、オリジナルの内容も書き換えられます。関数の中では、通常の配列と同じように添え字演算子[]を利用して目的の要素を指定しています。

次のプログラムは、ポインタとアロー演算子を利用して要素を指定し、内容を変更しています。

```
// kouTest17.c
typedef struct
{
    int year; // 学年
    int clas; // クラス
    int number; // 出席番号
    char name[64]; // 名前
    double height; // 身長
    double weight; // 体重
} seito;

void seito_print2(seito *pdata, int n);

int main(void)
{
    seito data[5]; // 5件の構造体を記憶する配列data[]
    int n;

    data->year = 3; // 「->」でアクセスしてみる
    data->clas = 4;
    data->number = 18;
    strcpy( data->name, "田中");
    data->height = 168.2;
    data->weight = 72.4;
    //
    (data + 1)->year = 2;
    (data + 1)->clas = 1;
    (data + 1)->number = 10;
    strcpy( (data + 1)->name, "佐々木");
    (data + 1)->height = 160.5;
    (data + 1)->weight = 58.5;
    // 3件目以降は省略

    n = 1;
    seito_print2(data, n); // 構造体の配列と表示する要素の番号を渡す

    printf("**** 関数を呼び出した後でデータを表示してみる ****\n");
    printf("[学年]:%d\n", (data + n)->year);
    printf("[クラス]:%d\n", (data + n)->clas);
    printf("[出席番号]:%d\n", (data + n)->number);
    printf("[名前]:%s\n", (data + n)->name);
    printf("[身長]:%f\n", (data + n)->height);
    printf("[体重]:%f\n", (data + n)->weight);

    return (0);
}

void seito_print2(seito *x, int n)
{
    (x + n)->height = 99999.9999; // 受け渡されたデータを変更してみる
    printf("**** 関数内でデータを書き換えてみる2 ****\n");
    printf("[学年]:%d\n", (x + n)->year);
    printf("[クラス]:%d\n", (x + n)->clas);
    printf("[出席番号]:%d\n", (x + n)->number);
    printf("[名前]:%s\n", (x + n)->name);
    printf("[身長]:%f\n", (x + n)->height);
    printf("[体重]:%f\n", (x + n)->weight);
    return;
}

*****実行結果*****
**** 関数内でデータを書き換えてみる2 ****
[学年]:2
[クラス]:1
[出席番号]:10
[名前]:佐々木
[身長]:99999.999900
[体重]:58.500000
```

~~~~~

\*\*\* 関数を呼び出した後でデータを表示してみる \*\*\*

[学年]:2

[クラス]:1

[出席番号]:10

[名前]:佐々木

[身長]:99999.999900

[体重]:58.500000

## 6. 構造体と初期化

構造体を初期化する場合は、配列と同じようにイニシャライザーを利用します。

struct タグ名 変数名 = {値1, 値2, ...};

このように記述することで、変数や配列を宣言すると同時に初期化できます。

※ANSI-Cに準拠していないCコンパイラの場合はエラーが出る場合があります。その場合は、先頭にstatic宣言を記述してください。

次のプログラムは、初期化子を利用して構造体の変数と構造体の配列を利用しています。

```
// kouTest18.c
// 構造体配列を初期化する
typedef struct
{
    int year; // 学年
    int clas; // クラス
    int number; // 出席番号
    char name[64]; // 名前
    double height; // 身長
    double weight; // 体重
} seito;

int main(void)
{
    int i;
    seito data1 = {1, 2, 18, "田中", 168.2, 72.4};
    seito data2[5] = {
        {3, 1, 101, "浅野", 170.2, 67.2},
        {3, 1, 102, "伊藤", 155.3, 70.3},
        {3, 2, 103, "鎌田", 167.2, 56.7},
        {3, 2, 103, "木村", 180.1, 50.6},
        {3, 3, 103, "佐藤", 157.1, 68.5},
    }; // 5件の構造体を記憶する配列data[]

    printf("**** 関数を呼び出した後でデータを表示してみる ****\n");
    printf("[学年]:%d, ", data1.year);
    printf("[クラス]:%d, ", data1.clas);
    printf("[出席番号]:%d, ", data1.number);
    printf("[名前]:%s, ", data1.name);
    printf("[身長]:%.1f, ", data1.height);
    printf("[体重]:%.1f\n", data1.weight);

    printf("\n");
    for(i = 0; i < 5; i++)
    {
        printf("[学年]:%d, ", data2[i].year);
        printf("[クラス]:%d, ", data2[i].clas);
        printf("[出席番号]:%d, ", data2[i].number);
        printf("[名前]:%s, ", data2[i].name);
        printf("[身長]:%.1f, ", data2[i].height);
        printf("[体重]:%.1f\n", data2[i].weight);
    }
    return (0);
}

*****実行結果*****
**** 関数を呼び出した後でデータを表示してみる ****
[学年]:1, [クラス]:2, [出席番号]:18, [名前]:田中, [身長]:168.2, [体重]:72.4

[学年]:3, [クラス]:1, [出席番号]:101, [名前]:浅野, [身長]:170.2, [体重]:67.2
[学年]:3, [クラス]:1, [出席番号]:102, [名前]:伊藤, [身長]:155.3, [体重]:70.3
[学年]:3, [クラス]:2, [出席番号]:103, [名前]:鎌田, [身長]:167.2, [体重]:56.7
[学年]:3, [クラス]:2, [出席番号]:103, [名前]:木村, [身長]:180.1, [体重]:50.6
[学年]:3, [クラス]:3, [出席番号]:103, [名前]:佐藤, [身長]:157.1, [体重]:68.5
```

以上