# 情報構造 第二回

データ型

# 今日の予定：データの話

- データ型
  - 原子データ型
  - 構造データ型
- 配列型
- 構造体型
- ポインタ型
  - ポインタに関するあれこれ

# データ型

- 機械語のデータ型
  - レジスタ値とその番地
- 原子データ型
  - char, int, long short…
- 構造データ型
  - 配列，構造体
- 基本的データ構造
  - リスト，スタック，キュー
- 先進的データ構造
  - 木，ハッシュ

機械語
型なし

昔の言語
C, Pascal

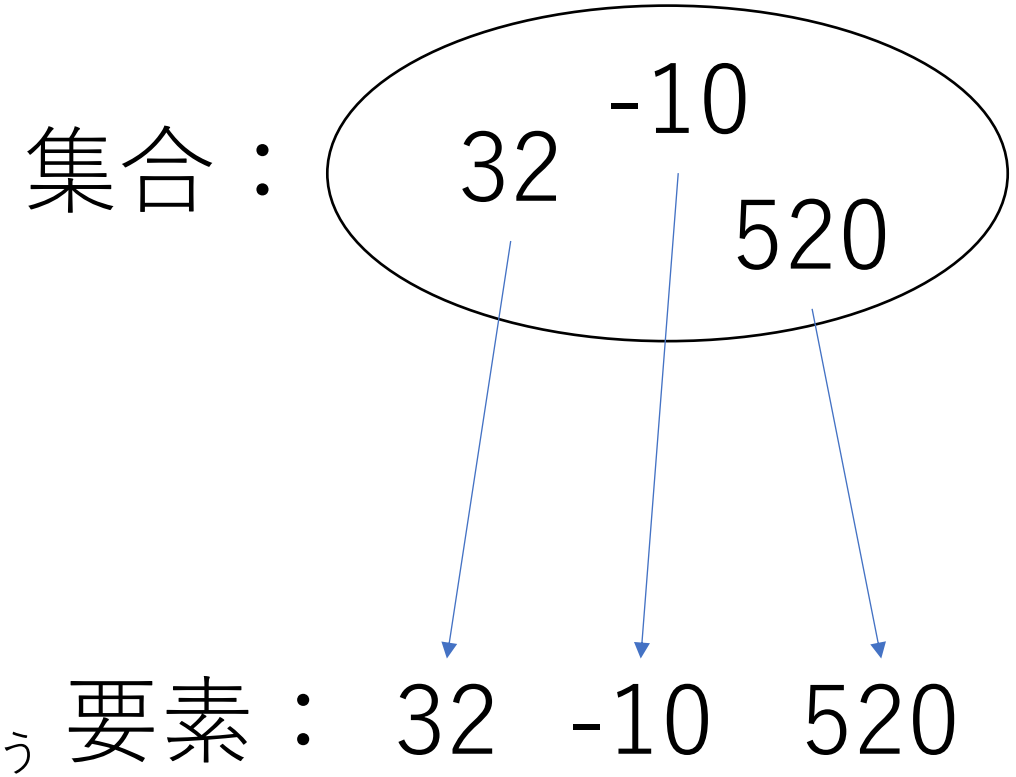
最近の言語
C++, Java…

# データ値 (data value)

- 例えば．．．

- 集合値：{32, -10, 520}
  - 要素（element）に分解できる
    - 分解できるデータ値を「構造データ値」という

- 整数値：592
  - これ以上分解ができない
    - 分解できないデータ値を「原子データ値」という




- どこまで分解できるかは，プログラミング言語に依存する

# データ型（data type）

- 同じ種類（例えば「整数」とか「実数」）のデータ値の集合
- これらを値の上で演算

- 例えば：整数型
  - 整数値の集合：0, 1, 2, …, -1, -2, …
  - 整数の演算子：+, -, *,…

# データ型のクラス

- 原子データ型
  - 値は分解不可能
    - 整数値，実数値など

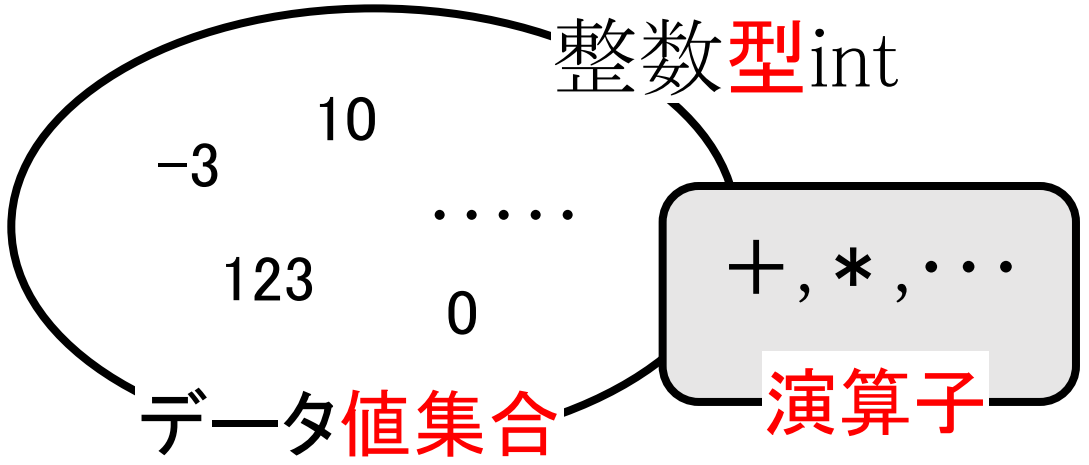
- 構造データ型
  - 値は分解可能
    - 集合型，配列型，構造体型など

# 原子データ型（atomic data type）

- 原子データ値
  - これ以上分解不可能なデータ値

- 原子データ型
  - 同じ種類の原子データ値の集まり
    ＋
  - それらを演算する演算子の集まり

  - 例えば：整数型，実数型，論理型など



# C言語の例：整数をあらわす原子データ型

- int, char, short, longのような**型指定子**によって整数値を表す
- データ値の集合：整数の部分集合
  - 整数を何byteで表すかによって型が異なる
- 演算子：
  - +, -, *, /, %（算術演算子）
  - ==, !=, <, >=など（関係演算子）
  - =（代入演算子）
  - など

| 型 | 説明 |
|---|---|
| int | 4byteまたは8byteの符号付き整数値（コンパイラに依存） |
| char | 1byteの符号付き整数（-128~127）（文字） |
| short | 4byteの符号付き整数（$2^{15}$~$2^{15}$-1） |
| long | 8byteの符号付き整数（$2^{31}$~$2^{31}$-1） |

# C言語の例：整数をあらわす原子データ型

- 宣言例：int i
- 整数値の例：10, -3, 23763, 0
- 演算の例:
  - 10+3（=13）, 123/10（=12）
  - 10<123（=1）, 4>=3（=0）

- 宣言例：char c
- 整数値の例：65 (‘A’) , 97 (‘a’)
- 演算の例：
  - 65 + 32 (= 97)
  - ‘A’ + 32 (= ‘a’)
  - char型は文字を表すための整数値

# 構造データ型（structured data type）

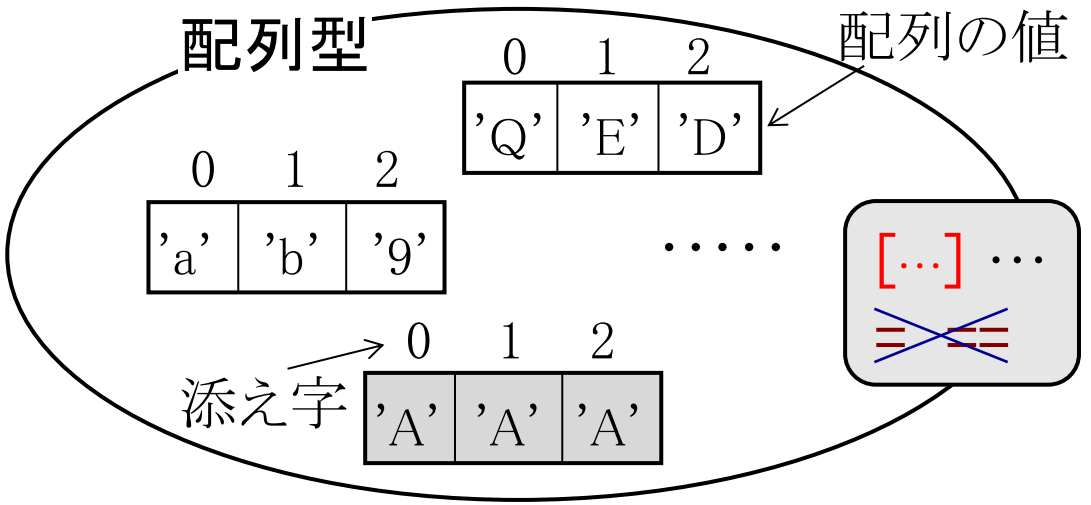
- データをより効率的に扱うための表現形式
- 要素を関係づける規則（構造）でつくられる

- 構造データ型はつぎの性質を持つ
  1. データ値が要素に分解できる
  2. 要素間の関係を表す構造を持つ
  3. データ値上の演算を持つ

- 例えば
  - 配列型：等しい型の要素から成り，それぞれの要素の位置を指定する添字をもつ構造
  - 構造体型（レコード型）
  - ポインタ型
  - さらに複雑な構造も．．．（データ構造（data structure））

# データ構造：配列型

- 多くの言語で用意されているデータ構造
  - C, Java, Pascal, FORTRAN, ALGOL, PL/I, Adaなど
- 配列型におけるデータ値の要素は**すべて同じ型**
- 各要素はその位置を指定する**添字**（index）を持つ
- 添字による要素指定は，**配列型の演算**である



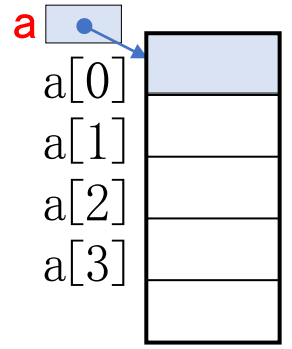
# C言語の配列

- 要素の型：char, 要素数：nの配列型 h の定義
  typedef char h[n];
  - nは定数にする必要あり（例えばh[10]）

typedef: データ型の宣言
typedefで，int型などもつくれる

```
typedef int aaa; // aaa型はint型
aaa a;
int b;  // aもbもint型
```

- データ値の集合：添字は0からn-1の文字型の要素の集まり
- 演算子：[…]で要素指定



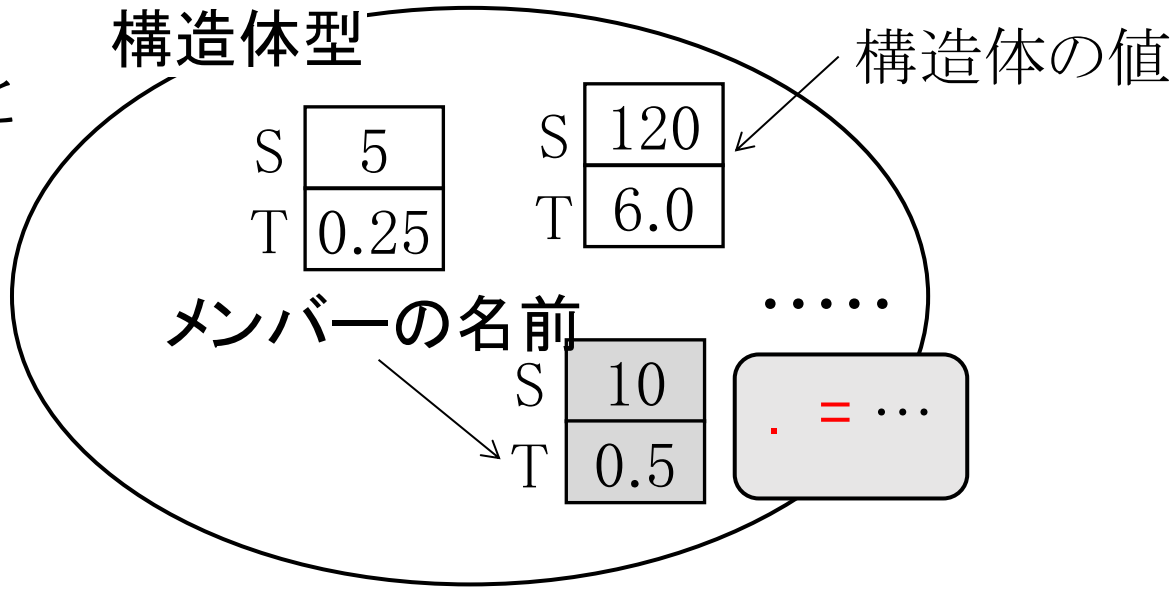

- 配列aの宣言

```
h a;
int i;
for ( i=0 ; i<3 ; i++) a[i]= ‘a’;
```

# データ構造：構造体型（レコード型）

- CやPL/Iで提供されているデータ構造
- ML, Pascal, Adaではレコード型と呼ばれる

- 構造体のデータ値の要素はメンバーと呼ばれる
- メンバーは異なった型を許す
- 各要素には名前がつけられ，この名前で要素を指定する



# C言語の構造体型

```
struct 構造体名{
    メンバ1;
    メンバ2;
    …
};
```

```
typedef struct 構造体名{
    メンバ1;
    メンバ2;
    …
} 構造体名2;
```

- typedef struct タグ名 { ～～ } 型名;
- メンバ名が S（整数型の要素）と T（実数型の要素）からなる構造体型 R の定義

```
typedef struct r {
        int S;
        float T;
} R;
```

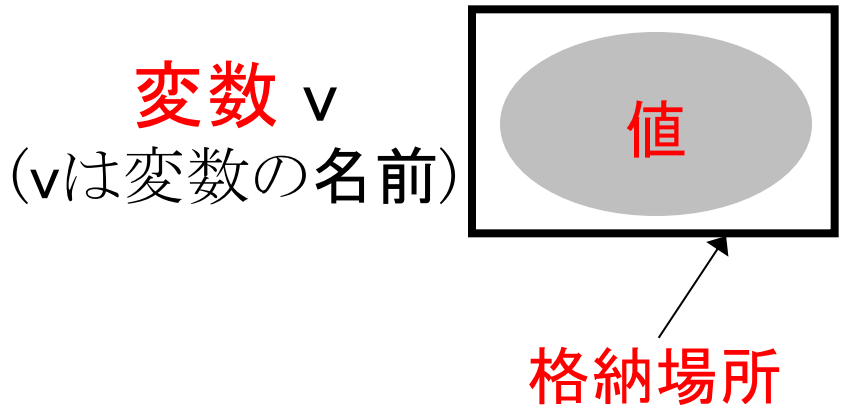
- データ値の集合：メンバー名Sの整数値の要素と，メンバー名Tの実数値の要素の集まり
- 演算子：
  - .（ピリオド）でメンバーにアクセス
  - ＝で代入

# C言語の構造体型（つづき）

- 宣言

```
R b, c;
b.S = 10;          // 演算子　.　メンバ変数へアクセス
b.T = 0.5
c = b;             // 演算子 = 構造体の代入
```

# データ構造：ポインタ型




- 変数
  - 名前：変数名
  - 中身：値
  - 入れ物：格納場所（メモリー上のアドレス）

- ポインタ変数
  - 名前：ポインタ変数名
  - 中身：格納場所

コンピュータのデータを格納する場所（メモリー）は有限
データ構造において，メモリーを意識することは有益

# C言語では

- 変数
  - 名前：v
  - 値：10
  - 格納場所（アドレス）：&v
- ポインタ変数（アドレスを格納する変数）
  - 名前: pv（宣言時は*pv）
  - 値：格納場所（アドレス）
  - 格納場所の値：*pv

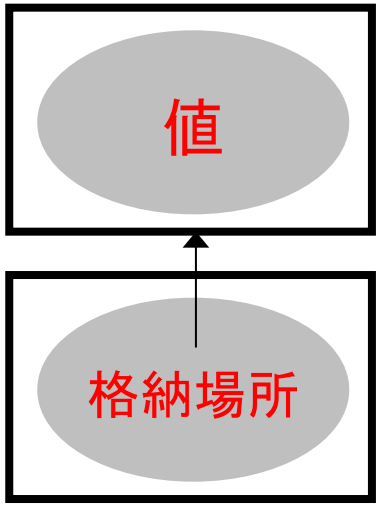
  &: アドレス演算子
  *: 間接演算子

```
int v;
v = 10;
printf("%d¥n", v);

int *pv;
pv = &v;
printf("%d¥n", *pv);
```

変数 v

ポインタ変数 pv

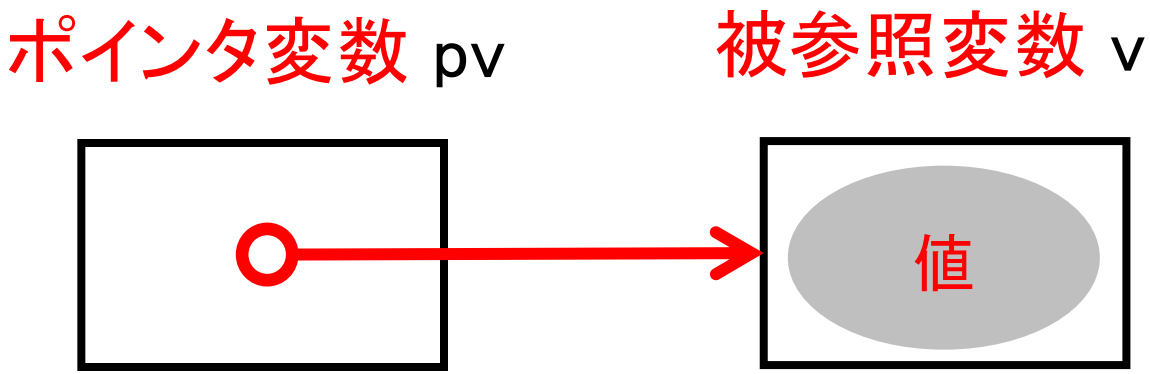



| メモリー上のアドレス | 値 | 変数名 |
|---|---|---|
| #0001 | 10 | v<br>*pv |
| #0002 | | |
| #0003 | | |
| #0004 | | |
| #0005 | #0001 | pv |
| #0006 | | |
| #0007 | | |
| #0008 | | |

# ポインタ：指す矢印

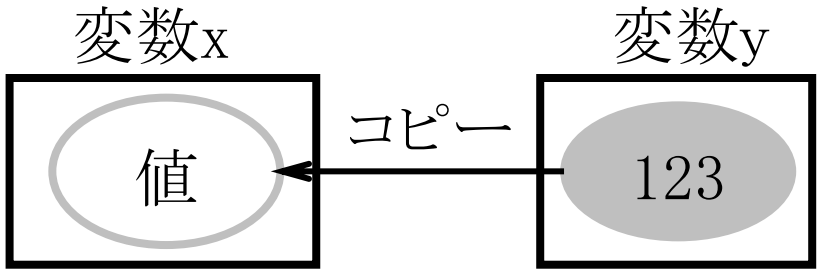
- ポインタは，別の変数を「指す矢印」で表示

- 指す：「参照する」とも言う
- 指される変数を被参照変数とも言う

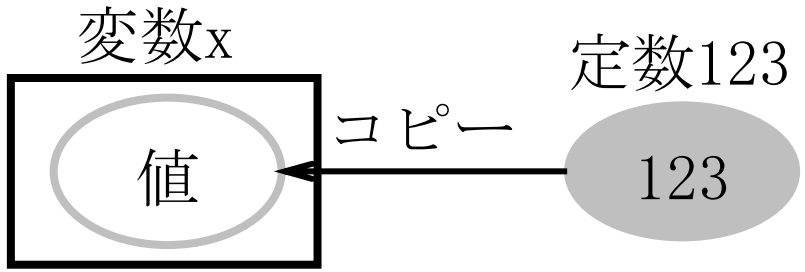


# 変数への代入: x = y を考える

- 右辺の変数 y の値をコピーして
- 左辺の変数 x の格納場所（メモリー）に入れる



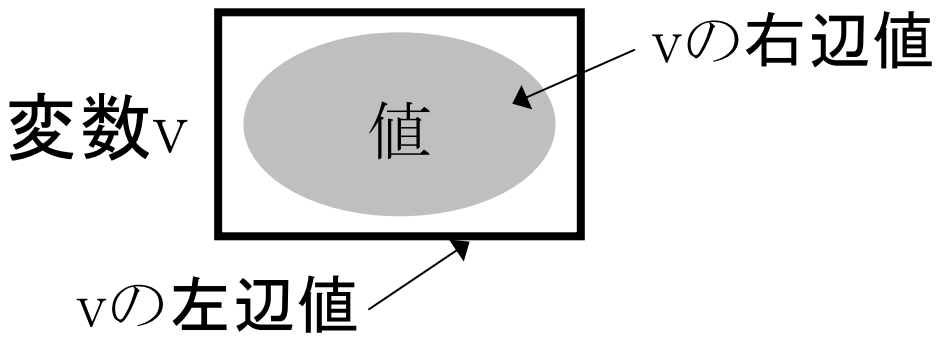

C代入文 x=y




C代入文 x=123

# 左辺値と右辺値

- 変数
  - 代入文 x = y の左辺 x とは
    - 入れ物（格納場所）=> 左辺値（l-value）という
  - 代入文 x = y の右辺 y とは
    - 中身（値）=> 右辺値（r-value）という



# 値呼び（値渡し）：call by value

- 手続きを呼び出すとき、
  ①**実引数**を評価し、
  ②**その**値を**仮引数**に渡す。

- 仮引数を実引数の値で**初期設定**する効果がある。
- 仮引数の値の変更は、実引数に及ばない！

# 値呼び（値渡し）：call by value

```
void f(long b){
    b += 100;
    printf(“%d¥n”, b);
}

main(){
    long a = 1000;
    f(a);

    printf(“%d¥n”, a); //1000
}
```

| メモリー上のアドレス | 値 | 変数名 |
|---|---|---|
| #0001 | 1000 | a |
| #0002 | | |
| #0003 | | |
| #0004 | | |
| #0005 | | |
| #0006 | | |
| #0007 | | |
| #0008 | | |
| #0009 | 1000 | b |
| #0010 | | |
| #0011 | | |
| #0012 | | |
| #0013 | | |
| #0014 | | |
| #0015 | | |
| #0016 | | |

# 参照呼び（参照渡し）：call by reference

- **番地呼び**(call-by-address)、**参照呼び**(call-by-reference)ともいう。
- 手続きを呼び出すとき、実引数の**番地**を仮引数に渡す。
- 呼び出された手続き内で**仮引数が引用**されたときは、この**番地を間接参照**して、**実引数**をアクセスする。

# 参照呼び（参照渡し）：call by reference

```
void f(long *p){
    *p += 100;
    printf(“%d¥n”, *p);
}

main(){
    long a = 1000;
    f(&a);

    printf(“%d¥n”, a); //1100
}
```

| メモリー上のアドレス | 値 | 変数名 |
|---|---|---|
| #0001 | 1000 | a |
| #0002 | | |
| #0003 | | |
| #0004 | | |
| #0005 | | |
| #0006 | | |
| #0007 | | |
| #0008 | | |
| #0009 | #0001 | p |
| #0010 | | |
| #0011 | | |
| #0012 | | |
| #0013 | | |
| #0014 | | |
| #0015 | | |
| #0016 | | |

参照呼びはメモリの節約

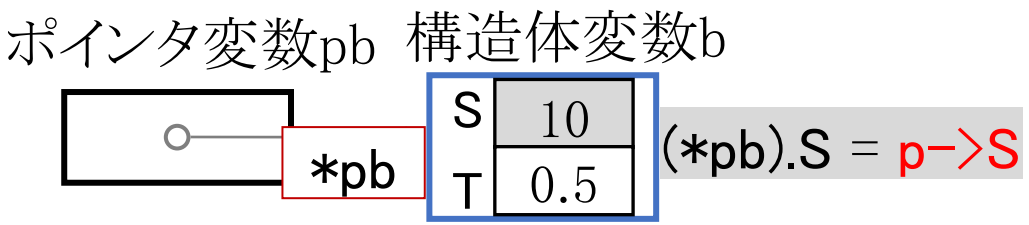
（最近のポインタは8バイトですが．．．）

# 構造体の参照（アロー演算子）

```
typedef struct r {
    int S;
    float T;
} R;
```

- において

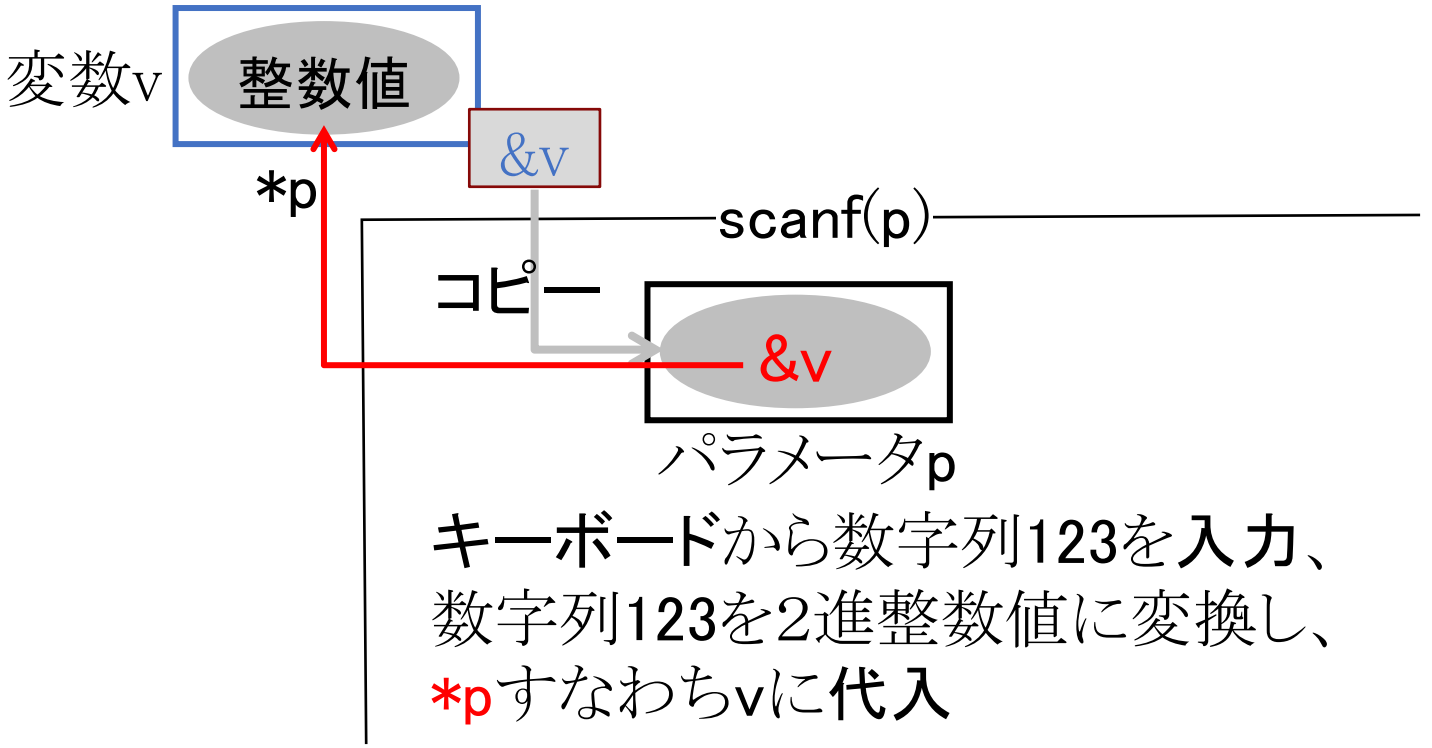
```
R b, *pb;
b.S = 10;
b.T = 0.5;
pb = &b;
printf("S: %d T: %f¥n", (*pb).S, (*pb).T);
printf("S: %d T: %f¥n", pb->S, pb->T);
```




ポインタから構造体メンバの値へ
直接アクセス

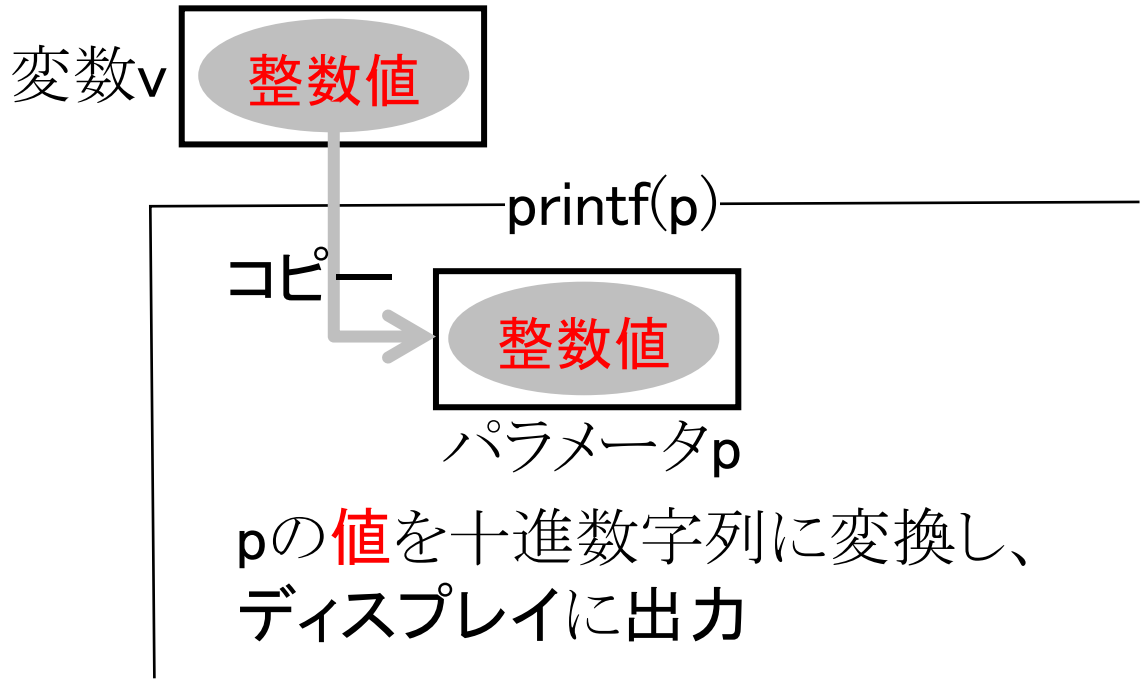
# C言語の入力関数：ポインタ操作例

- **入力関数での変数指定**
  int v のとき
  scanf(″%d″, &v)
  　&v（変数vの番地）を指定。



# C言語の出力関数

- **出力関数での変数指定**
  int v のとき
  printf(″%d″, v)
  　vの右辺値（変数vの値）を出力する



# 共用体（union）

- 構造体と似ているが,
- 共用体は一つのメモリに複数のメンバ変数を割り当てる

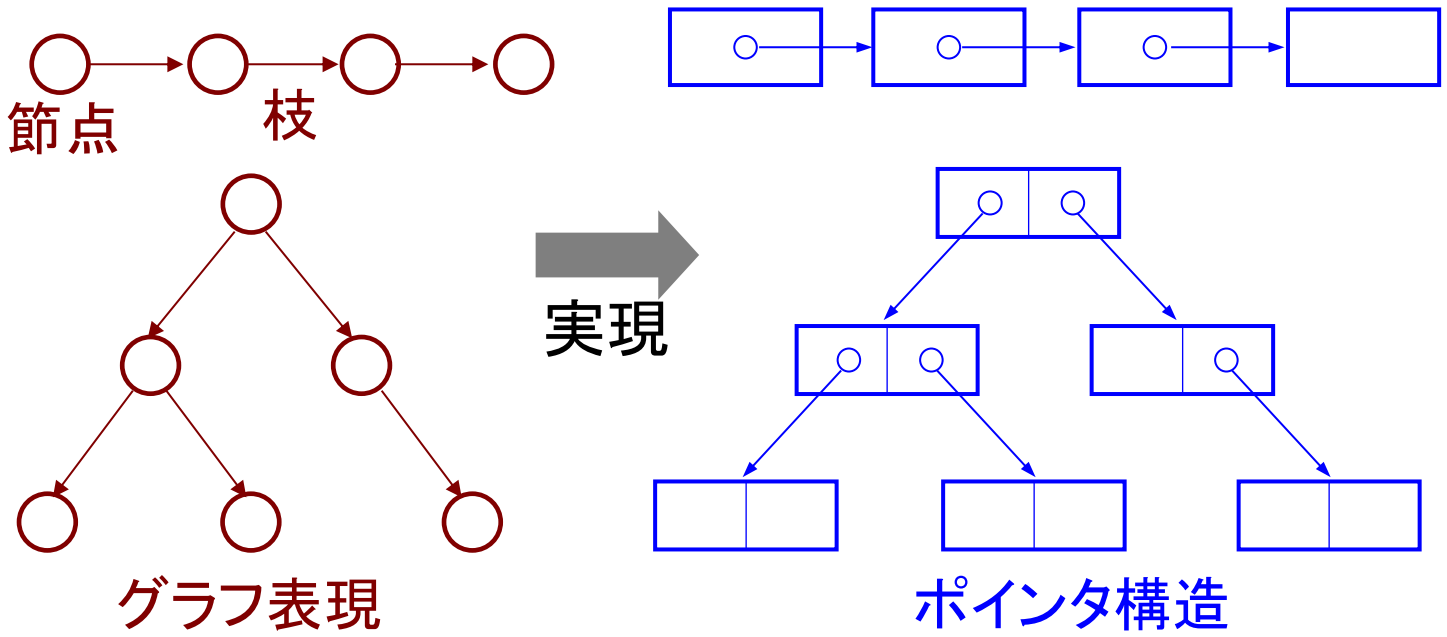
```
typedef union u{
    int S;
    float T;
}U;

U a;
a.S = 5;
printf(“S: %d T: %f”, a.S, a.T);

a.T = 0.5;                          // a.Sのメモリに上書きされる
printf(“S: %d T: %f”, a.S, a.T);
```

# グラフのポインタ表現

- **グラフ**…関係を記述する有効な表現
- グラフの**節点**（**頂点**） ⇒ **ポインタ変数**や**被参照変数**
- グラフの**枝**（**辺**） ⇒ **ポインタ**
- **グラフ**は**ポインタ構造**で表現できる



# C言語の記憶領域の確保と解放

- 動的記憶領域（ヒープ）から被参照変数の割り当て

  p = (T *)malloc( sizeof( T ) );

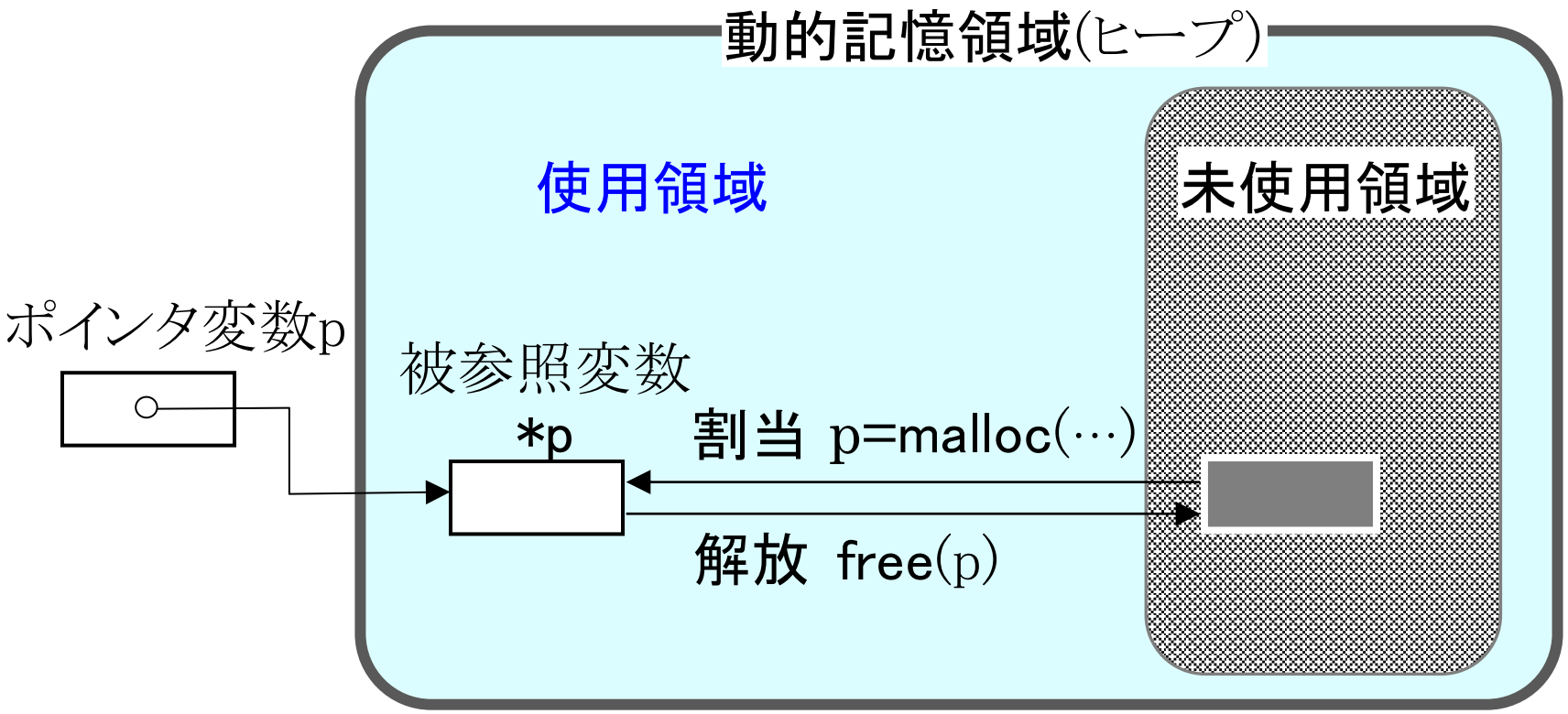
  - sizeofは使用するメモリー使用量（型Tのメモリー使用量）
  - データ型Tの変数を動的記憶領域（ヒープ）上での割り当て

# 動的変数，動的記憶領域

- 扱うデータの大きさが終了時まで決まっている場合：静的記憶領域
- 実行時にデータの扱うデータの大きさが決まる：動的記憶領域（ヒープ）
  - 実行中に必要な分だけメモリを確保（malloc）し，
  - 使い終わったら解放（free）する
  - このような変数を動的変数という
- 動的変数は名前がないので，ポインタ変数でアクセスする

```
T *p;                                  // T型のポインタ変数pを宣言
p = (T*) malloc( sizeof( T ) ); // T型のメモリーサイズのヒープを確保しそのアドレスをpに代入
free( p );                             // ヒープのメモリーを解放
```

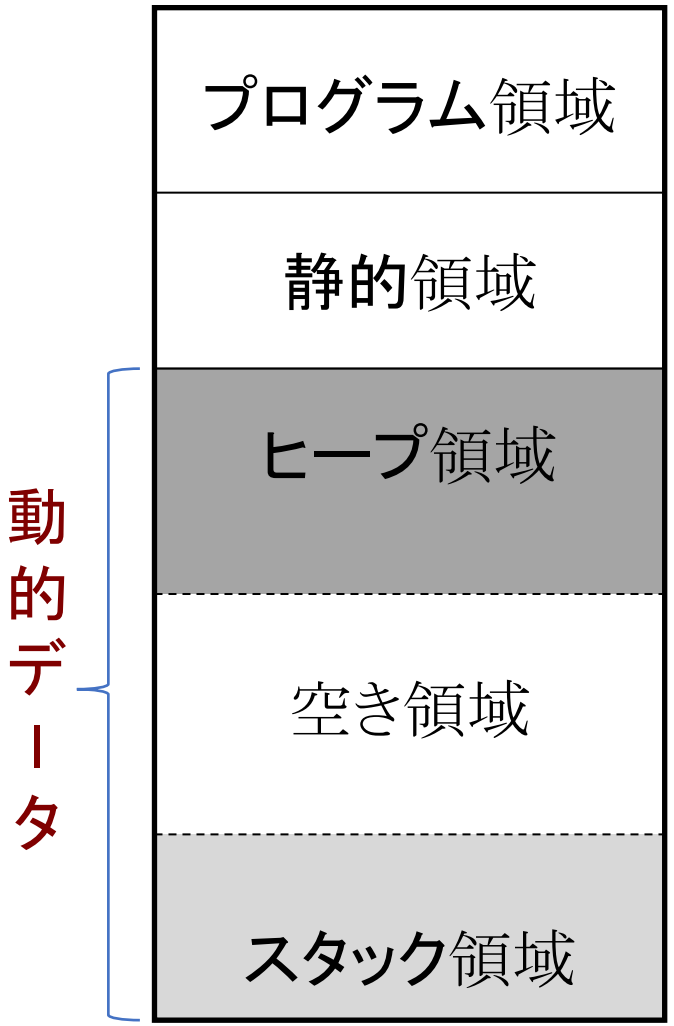
# 動的記憶領域 (ヒープ)



# 自動変数，スタック領域

- 実行中に自動にメモリー領域を確保したり開放したりする領域をスタック領域という
  - ブロック（例えば関数）の実行開始
    - => 必要なメモリー（例えばローカル変数）をスタック領域から確保
  - ブロックの実行終了
    - =>スタック領域から解放
- 宣言
  - auto int a;　　// 明示的
  - int a;　　// autoは省略される

# C言語の記憶領域（処理系によって異なる）




プログラム実行のためのプログラムコード

静的オブジェクト
(グローバル変数や，static宣言された変数)

mallocで割り当てられたオブジェクト

自動オブジェクト
(auto宣言されたもの)

# まとめ

- データ構造
  - 原始データ型と構造データ型
  - 配列型
  - 構造体型
  - ポインタ型

# 演習

- 1. 構造体と共用体
  - 構造体をつくってみる
  - 共用体をつくってみる
  - 構造体と共用体の差を確認する
- 2. ポインタ
  - 値呼び（値渡し）と参照呼び（参照渡し）の関数を作ってみる
  - ポインタ変数とピリオドで，構造体の要素を参照する
  - アロー演算子で，構造体の参照を行ってみる

- 最低限授業で示したプログラムは試して下さい.
- 自分で作って試してもOK（その場合は授業のプログラムは必要なし）

- 作成したソースコードをletusで提出して下さい
- 締切：5/1（月）10:30